AFRL-AFOSR-UK-TR-2014-0042

Dynamic Information Management and Exchange for Command and Control Applications

Barbara Carminati
Elena Ferrari

UNIVERSITA' DEGLI STUDI DELL'INSUBRIA
DIPARTIMENTO DI INFORMATICA E COMUNICAZIONE
VIA GIUSEPPE MAZZINI 5, VARESE 21100 ITALY

EOARD Grant 10-3080

Report Date: March 2014

Final Report from 24 August 2010 to 12 March 2014

Air Force Research Laboratory
Air Force Office of Scientific Research
European Office of Aerospace Research and Development
Unit 4515, APO AE 09421-4515

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From – To)*
12 Mar 2014	Final Report	08/24/2010 – 03/12/2014

4. TITLE AND SUBTITLE

Dynamic Information Management and Exchange for Command and Control Applications

5a. CONTRACT NUMBER

FA8655-10-1-3080

5b. GRANT NUMBER

Grant 10-3080

5c. PROGRAM ELEMENT NUMBER

61102F

6. AUTHOR(S)

Barbara Carminati
Elena Ferrari

5d. PROJECT NUMBER

5d. TASK NUMBER

5e. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
UNIVERSITA' DEGLI STUDI DELL'INSUBRIA
DIPARTIMENTO DI INFORMATICA E COMUNICAZIONE
VIA GIUSEPPE MAZZINI 5, VARESE 21100 ITALY

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

EOARD
Unit 4515
APO AE 09421-4515

10. SPONSOR/MONITOR'S ACRONYM(S)

AFRL/AFOSR/IOE (EOARD)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

AFRL-AFOSR-UK-TR-2014-0042

12. DISTRIBUTION/AVAILABILITY STATEMENT

Distribution A: Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

In traditional information systems, information sharing is usually regulated by a proper set of pre-defined access control policies that state who can access which data portion and under which mode. This model does not fit the emergency management scenario, where usually there is the need to access resources with greater flexibility than during the normal system operations. In order to achieve this goal, when an emergency happens, it is necessary to bypass the regular access control policies and grant users access to resources not normally authorized, in order to handle the emergency. However, such downgrading of the information security classification should be temporary and controlled, that is, regulated by proper emergency policies. In this report we describe a proposed access control framework in support of emergency management.

15. SUBJECT TERMS

EOARD, Information Management, Distributed Computing, Information Assurance

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18, NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			James H Lawton, PhD
UNCLAS	UNCLAS	UNCLAS	SAR	157	**19b. TELEPHONE NUMBER** *(Include area code)* (703)696-5999

Dynamic Information Management and Exchange for Command and Control Applications

Grant FA8655-10-1-3080

Final Report: "A Framework in Support of Emergency Management for Specified and Unspecified Emergencies"

Barbara Carminati (PI), Elena Ferrari (co-PI)

University of Insubria

A report submitted to the European Office of Aerospace Research & Development

Distribution A: Approved for public release; distribution is unlimited.

Contents

Distribution A: Approved for public release; distribution is unlimited.

In the last years, natural catastrophic events, e.g., floods, earthquakes, hurricanes [2] and man-made disasters, e.g., airplane crashes, terrorist attacks, nuclear accidents [7], highlight the need for a more efficient emergency management. In particular, attacks of September 11, 2001, have shown that the lack of effective *information sharing* resulted in the failure to intercept the terrorist attacks [1]. This example points out the need of a more efficient, timely and flexible information sharing during emergency management.

In traditional information systems, information sharing is usually regulated by a proper set of pre-defined access control policies that state who can access which data portion and under which mode [48]. This model does not fit the emergency management scenario, where usually there is the need to access resources with greater flexibility than during the normal system operations. In order to achieve this goal, when an emergency happens, it is necessary to bypass the regular access control policies and grant users access to resources not normally authorized, in order to handle the emergency. However, such downgrading of the information security classification should be *temporary* and *controlled*, that is, regulated by proper *emergency policies.* In this report we describe the proposed framework in support of emergency management.

Chapter 1

Introduction

In the last years, natural catastrophic events, e.g., floods, earthquakes, hurricanes [2] and man-made disasters, e.g., airplane crashes, terrorist attacks, nuclear accidents [7], highlight the need for a more efficient emergency management. In particular, attacks of September 11, 2001, have shown that the lack of effective *information sharing* resulted in the failure to intercept the terrorist attacks [1]. This example points out the need of a more efficient, timely and flexible information sharing during emergency management.

In traditional information systems, information sharing is usually regulated by a proper set of pre-defined access control policies that state who can access which data portion and under which mode [48]. This model does not fit the emergency management scenario, where usually there is the need to access resources with greater flexibility than during the normal system operations. In order to achieve this goal, when an emergency happens, it is necessary to bypass the regular access control policies and grant users access to resources not normally authorized, in order to handle the emergency. However, such downgrading of the information security classification should be *temporary* and *controlled*, that is, regulated by proper *emergency policies*.

To cope with these requirements in [27], we propose an access control model to enforce controlled information sharing in emergency situations. Our model is able to enforce flexible information sharing through the specification and enforcement of *emergency policies*. Emergency policies allow the instantiation of *temporary access control policies* that override regular policies during emergency situations. More precisely, each emergency is associated with one or more temporary access control policy templates, describing the new access rights to be enforced during specific emergency situations. *Emergency obligations* are also supported, since the detection of an emergency could require the immediate execution of some activities encoded according to a given response plan.

The *core emergency policy* model proposed in [27] has been extended in order to support composed emergency policies [28] and administration policies [31].

The former extension [28] introduces the concept of *composed emergencies*, to describe how *atomic emergencies* can be combined together to from a composed one. Moreover, in some cases, a composed emergency may require overriding the tacps/obligations that have been activated as response plans of its sub-emergencies, whereas in other cases tacps/obligations of composed emergencies should coexist with those of its sub-emergencies. Therefore, we associate with each policy for composed emergencies an *overriding strategy* according to which we can specify if tacps/obligations of sub-emergencies have to be maintained, deleted or temporarily blocked until the end of the composed emergency.

The latter extension [31] concerns *administration policies.* Emergency management is a complex task which requires distributing the rights of create/modify emergency policies among different subjects. In order to distribute these rights, we introduce administration policies that specify who are people authorized to create/modify policies and which emergencies and polices they are authorized to specify. These restrictions are captured through the definition of proper *scopes* that limit the right to state emergency policies within specific constraints.

1.1 Unspecified Emergency Management

The *core emergency policy* model (and its extensions) is able to deal only with emergency situations which can be specified *a-priori.* In many domains this is enough since, in emergency management, it is common that experts of the field define response plans, based on regulations and laws as well as on reports resulting by the emergency preparedness and risk assessment phase [47]. All these documents represent a solid base from which emergencies and emergency policies can be specified. However, there are many scenarios where this might be not enough, since it is difficult to *a-priori* figure out all possible emergency situations. For instance, in the healthcare domain, it is difficult to model a priori any possible disease or injury which might be considered as an emergency and associate with it the correct information need. This may have serious consequences, in that *unspecified emergencies* are not covered by defined policies and therefore the system is not able to respond to their information needs, unless someone manually triggers the emergency status. This is the idea behind the break-the-glass model (cfr. Chapter 2 for more details). However, we believe that the risk of information leakage caused by let the user arbitrarily breaking the glass might seriously impact the system security. For this reason, we explore an alternative approach to deal with unspecified emergencies highly reducing the risk of information leakage.

The basic idea is to open the system to some *access control violations*, i.e., to grant access to some requests that normally should be denied, but that may be permitted due to the happening of an unspecified emergency. The reason behind this choice is that the risk of data leakage generated by these violations might be lower than the damage caused by a late emergency response. Let us consider, as an example, a power plant; usually only power plant technicians are allowed to access the schema of the infrastructure but, suppose that a firefighter Y requires the access to this information. If this access is denied, we might have that: (1) an unspecified emergency such as a fire alarm is really going on in the power plant, or (2) the request is an attempted abuse from Y. Definitely, in this latter case granting the access is a violation to the infrastructure schema confidentiality. However, we might agree that this damage is negligible compared to the benefits of granting the access in case of an unspecified fire emergency. Obviously, not all denied access requests have to be allowed. In contrast, the idea is to have a system open only to those denied access requests that have been blocked due to the absence of proper emergency policies. The problem is how to detect whether a denied access request is related to an unspecified emergency or it is simply an attempted abuse.

In order to achieve this goal, the *core emergency policy* model has been extended using three strategies for the management of unspecified emergencies: the *policy based analysis*, *anomaly based analysis* and *historical based analysis*. The anomaly based analysis combines anomaly detection techniques and complex event processing (CEP) in order to detect anomalous events which might represent unspecified emergencies and correlate these events to denied access requests. The historical based analysis considers previously permitted access requests in order to detect if the current access request is similar to one of them. For each of these strategies, we define measures called *satisfaction level*, *anomaly level* and *historical level*. These levels measure, respectively, how much an access request is close to satisfy existing policies, how much a set of events is anomalous w.r.t. the normal behavior, how much an access request is similar to the previously permitted access requests. Every time an access request is denied due to the absence of a proper policy, we exploit our strategies to recognize whether it represents an attempted abuse or an information need for an unspecified emergency. In the former case, the access request is denied, otherwise it is authorized as a controlled violation.

1.2 Organization

This report is organized as follows:

- Chapter 2 analyzes the state of the art of topics related to the access control model presented in this report. More precisely, the literature about Complex Event Processing (CEP) technology, flexible access control models based on Break the Glass (BtG) policies, context-based access control, obligations, policy composition, administrative access control, policy similarity and anomaly detection are presented.

- Chapter 3 presents the *core emergency policy* model for flexible information sharing in emergency management. The following extensions of the core model are also presented: (i) policy composition, (ii) administrative emergency policies, (iii) correctness validity checks.

- Chapter 4 presents the extension of the core model which supports for unplanned emergency based on access request analysis and anomaly detection.

- In Chapter 5, a prototype implementation of the proposed access control model is explained. The prototype architecture details are provided with a full explanation of the technologies used in the implementation and the functionalities of the prototype are explained.

- Chapter 6 shows experimental results of a wide series of tests performed on the prototype. A set of test on the core model prototype performance are provided and a series of experiments on correctness of the extension on unspecified emergency is presented.

- Chapter 7 draws some conclusions summarizing the main results and discussing plans for future works.

1.3 Publications

The report is based on Ph.D Thesis of Dr Michele Guglielmi, and part of the material here presented has already been published, as described in the following.

- The core emergency policy model for flexible emergency management has been published in [27].

- The extension of the core model for the support of composed emergency policies has been presented in [28].

- An in-depth analysis of the model is discussed in [31] and administration policies are introduced to enhance the model flexibility during emergencies.

- The prototype framework called SHARE (Secure information sHaring frAmework for emeRgency managemEnt) enforcing the core emergency policy model for emergency situations has been shown in [30].

- The extended version of the access control model able to deal with unspecified emergencies has been presented in [29, 26].

Chapter 2

State of the Art

The access control model proposed in this reportreport covers a large number of research areas. First of all, emergency detection is performed exploiting Complex Event Processing (CEP) technology (see Section 2.1). The emergency policy model is a flexible access control model similar to Break the Glass solutions (see Section 2.2), but with some important improvements. The proposed access control model is context-based (see Section 2.3) and makes use of obligations (see Section 2.4).

The core access control model has been extended in several directions: first of all policy composition has been introduced (see Section 2.5), then administrative access control has been added (see Section 2.6). The most important extension regards the management of unspecified emergencies. We focus on unspecified emergencies that are similar to emergency situations already registered in the system, thus we have defined measures to represent how close an access request is to satisfy existing policies. These measures have already been studied in the field of policy similarity (see Section 2.7). Finally, the last extension regards the automatic detection of emergency situations exploiting anomaly detection techniques (see Section 2.8).

2.1 Complex Event Processing

Emergency detection is major concern of this report. It can benefit from the recent advent of Complex Event Processing (CEP) systems [63] as they allow capturing complex event patterns signaling the beginning/ending of an emergency. CEP systems are able to continuously process flowing data from geographically distributed sources. CEP systems represent an evolution of Data Stream Management Systems (DSMSs): DSMSs process incoming data through a sequence of transformations based on common SQL operators to produce streams of new data as an output, whereas CEPs see incoming data as events happened in the external world, which have to be filtered and combined to detect occurrences of particular patterns.

The literature offers several languages for event pattern specification (e.g., Amit [4], XChange^{EQ} [43], SpaTec [73], TESLA [37] and SASE+ [5, 51]). Some languages have also been proposed by vendors (e.g., Streambase, Sybase, Oracle CEP). These languages mainly differ in the set of constructs they support. However, up to now, a standard event specification language has not yet emerged. To overcome the problem of lack of standardization, in the report a Core Event Specification Language (CESL) will be used.

CESL supports the following operators: (i) query stream operators such as *selection*, *projection*, *aggregation*, and *join*; (ii) basic event operators, like *event type*, *event instance* and *array of event instances*; and (iii) complex event pattern operators, such as *sequence*, *negation* and *iteration* of events. More details about CESL are provided in Section 3.1.

2.2 Break the Glass Policies

Traditional access control models are usually strict models where permissions are known in advance, but in real settings, unplanned emergency situations may occur. In these cases, a more flexible and adaptable approach can be adopted. Break-the-glass (BtG), introduced in [74], is an approach for such flexible support of policies which helps to prevent system stagnation that could harm lives or otherwise result in losses.

A BtG policy allows a user to override regular access control policies on demand. Usually, the usage of BtG policies needs to be documented for later audits and reviews. Several works have been done in the last years about BtG. Ferreira et al. [50, 49] presented a first approach to BtG based on *special accounts* that are temporary accounts that comprise more powerful access rights with a more detailed logging.

Another BtG model [22, 23], presented by Brucker and Petritsch, is based on *emergency levels*, i.e. policies are classified according to different levels (normal / low / medium / high emergency level) and active policies are normal or emergency policies whether the current system state is normal or under emergency.

Ardagna et all.[8] proposed an advanced approach to BtG based on the definition of different *policy spaces*, i.e. spaces for regular policies and for exception policies; when an access is not explicitly denied or permitted by a regular policy, the system checks exception policies and if there is an already planned exception the access is granted, otherwise if the exception is unplanned the system denies or permits the request whether the global status is normal or critic.

2.2.1 Comparison between BtG and Emergency Policies

In BtG models when an access request is denied, the system verifies whether this decision can be overridden by a BtG policy and, in such a case, the subject is notified and asked to confirm. In our proposal, when an access is denied by a regular policy, the system checks if this decision can be overridden by a temporary access control policy activated by the emergency and, in this case, the access is granted. The two approaches seem similar, but there are significant differences as explained Table 2.1.

BtG Model	Our Proposal
BtG policies are always active.	Emergency policies are active only during emergencies.
Users decide when override a regular policy.	The system automatically recognizes when an emergency occurs and overrides regular policies with emergency policies.
A user can arbitrarily break the glass.	Only the system can override a regular policy and only during an emergency.
A user might wait a while to respond when the system prompt the BtG request.	The system immediately overrides regular policies when an emergency is detected.

Table 2.1: Comparison between BtG and our proposal

Beside the points presented in Table 2.1 before it is worth noting some issues about security and flexibility.

- BtG models are more flexible because emergency policies allow violations only for specified emergencies, whereas BtG models manage also unspecified emergencies.
- BtG models are less secure because abuses of BtG accesses could bring the system to an unsafe state, whereas emergency policies do not allow violations thus abuses are not possible.

As we claimed in the introduction, our model is enough flexible for most of emergency management scenarios, since emergency policies can be defined based on risk assessment documents [47]. However, in some scenarios such as healthcare domain, this might be not enough, since it is difficult to figure out *a-priori* all possible emergency situations. Since unspecified emergencies are not covered by emergency polices, we extend our model in order to allow for controlled violations, but in a safer manner w.r.t. BtG policy models, i.e., highly reducing the risk of information leakage. The basic idea is to open the system to some *access control violations*, i.e., to grant access to some requests that normally should be denied, but that may be permitted due to the happening of an unspecified emergency. Even if this extension supports controlled violations, a relevant difference holds w.r.t. BtG models. In the proposed approach, violations are decided only by the system and not by requestor and only if the attempted access is close to satisfy one of existing policies. This makes the proposal safer than BtG models.

2.3 Context-based Access Control

Our model makes a large use of contextual information, not only in emergency description, but also to describe emergency contexts and access control context. Several works have studied how to model the concept of context for access control. Some works model specific context like temporal context [12] or spatial context [38], other works model generic context like Generalized Role-Based Access Control [65]. Context-aware access control can be realized mainly in two ways: *context constraint* or *conditions* and *context* or *environment roles*. A *context constraint* is defined by Strembeck et al. [76] as a dynamic constraint that checks the actual values of the context attributes captured from the environment; when a user performs an access request the system checks the associated context conditions and decides if the corresponding access can be granted. An *environment role*, as proposed by Covington et al. [35], associates environmental conditions with a role, i.e. an environment role can be *"weekdays"*, which means that this role is active only during weekdays; when a user performs an access request for a resource R the system checks the current active roles (included environmental roles) for the user, if the set of roles needed to access R is a subset of the user active roles then the access is granted. In our model, we use the Boolean condition abstraction to express a context constraint, but in a future implementation both of the previously presented approaches might be used (context constraints or environment roles).

Another work which is worth mentioning is the *category based access control model* presented by Barker et al. in [9, 15]. More precisely, this is a meta-model based on term rewriting. The meta-model is not only able to deal with contextual information, but also to express a wide range of traditional access control models. The model is based on the concept of *category*, i.e., classes or groups to which entities may be assigned. Entities are denoted uniquely by constants which include categories $\mathcal{C}$, principals $\mathcal{P}$, actions $\mathcal{A}$, resources $\mathcal{R}$ and events $\mathcal{E}$. Moreover, the meta-model defines relationships among entities such as principal-category assignment, permission and authorization.

- **Principal-category assignment:** $\mathcal{PCA} \subseteq \mathcal{P} \times \mathcal{C}$, such that $(p, c) \in \mathcal{PCA}$ iff a principal $p \in \mathcal{P}$ is assigned to the category $c \in \mathcal{C}$.
- **Permissions:** $\mathcal{ARCA} \subseteq \mathcal{A} \times \mathcal{R} \times \mathcal{C}$, such that $(a, r, c) \in \mathcal{ARCA}$ iff the action $a \in \mathcal{A}$ on resource $r \in \mathcal{R}$ can be performed by principals assigned to the category $c \in \mathcal{C}$.
- **Authorization:** $\mathcal{PAR} \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{R}$, such that $(p, a, r) \in \mathcal{PAR}$ iff a principal $p \in \mathcal{P}$ can perform the action $a \in \mathcal{A}$ on the resource $r \in \mathcal{R}$.

We believe that the meta-model presented in [9, 15] is able to express our emergency policy model. More precisely, a user role assignment (u, r) can be expressed as a principal-category assignment (p, c), where category c corresponds to role r and principal p corresponds to the user u.

A temporary access control policy $t = (r, o, p)$ which authorizes users belonging to role r to exercise the privilege p on objects identified by object specification o, can be expressed as a permission (a, r, c) where action a corresponds to privilege p, resource r corresponds to object o and category c corresponds to role r.

An authorization (u, p, o) which authorizes user u to exercise privilege p on a target object o can be expressed as an authorization (p, a, r) where principal p correspond to user u, action a correspond to privilege p and resource r correspond to object o.

The concept of emergency can be modeled as a category, i.e., during emergency e a principal is assigned to the corresponding category c_e.

Finally, the concept of emergency policy (e, t) which activates temporary access control policy t during emergency e can be expressed as a principal-category assignment (p, c_e) and a permission (a, r, c_e), in this case during emergency e, principal p is assigned to category c_e, thus it is authorized to execute action a on resource r.

2.4 Obligations

In order to handle an emergency situation, sometimes overriding an access control policies is not enough, because there is the need to perform certain actions to manage the emergency. To model these actions we use the concept of obligations. An *obligation* is an action or a set of actions that must be performed by system or users when certain events occur. When events are access control decisions they are called *access control obligations*. These kinds of obligations were analyzed by Bettini et al. in [17]. They formalized policies with *obligations* and *provisions*, allowing policies to specify actions and conditions to be fulfilled before or after user exercising of the granted privileges. Obligations were further extended in *system obligation* and *user obligation* whether the actions are fulfilled by the system or the user. In [16], Bettini et al describe how to monitoring user obligations in order to place under control the handling of user *obligation violations*. Working in the context of user obligations, Irwin et al. [52, 53] observe that deadlines are needed for user obligations in order to be able to capture the notion of violation of obligations. Several policy languages have been proposed that support the specification of obligations in security policies. Modern access control languages such as XACML [18] (and similarly, EPAL [69]) have limited models of obligations. Specifically, they model system obligations and cannot describe user obligations. PONDER language [39], and Policy Description Language (PDL) [62] support the specification of user obligations. In PONDER obligation policies specify the actions that must be performed by system or users when certain events occur. Obligation policies are event-triggered and define the activities subjects (human or automated manager components) must perform on objects in the target domain. Similarly, PDL policies use the event-condition-action rule paradigm of active databases to define a policy as a function that maps a series of events into a set of actions. These languages define a concept of obligations related to generic event and not necessarily linked to access control. These kinds of *event-triggered obligations* are very similar to our idea of emergency obligation policy. An emergency obligation policy is an event-triggered obligation where the event represents an emergency. For example, an obligation related to a cardiac arrest emergency, might automatically call the ambulance for the patient under emergency.

2.5 Policy Composition

Since our proposal deals with emergency policy composition, it may be considered in some relationship with work for policy composition [20, 6, 45, 14, 24, 25, 58, 67].

One early work on policy composition is the policy algebra proposed by Bonatti et al. [20], which aims at combining authorization specifications originating from heterogeneous independent parties. They model an access control policy as a set of ground (variablefree) authorization terms, where an authorization term is a triple of the form (subject, object, action).

Most recently, Bruns et al. [24, 25] proposed an algebra for fourvalued policies based on Belnap bilattice. In particular, they map four possible policy decisions, i.e. grant, deny, conflict and unspecified, to Belnap bilattice and claim that their algebra is complete and minimal.

However, it is important to note that the focus of our work is different from the scope of the proposals for policy composition, since we deal with composition of emergencies, to which we associate new emergency policies. As such, while policy combination strategies focus on operators to combine policies and resolution strategies for conflicts among positive and negative policies, in our proposal we are interested in composition of emergencies and solutions for the overriding or coexistence of their corresponding temporary access control policies, as required by the new response plan.

2.6 Administrative Access Control

Since emergency policies might be large in number, we believe the right to create/modify these polices should be distributed to multiple subjects called emergency managers. In order to distribute these rights without losing the central control by the administrator, we have defined administrative polices. The problem of administrative access control has been widely analyzed in literature. For instance, ARBAC97 [72] is the first work to specify an administrative access control model. ARBAC97 defines a set of administrative roles, which is disjoint from the set of normal roles. Only members of these roles can perform administrative operations.

Administrative access control has been also considered by the OASIS technical committee which has published the XACML v3.0 administration and delegation profile (XACML-Admin) working draft on 16 April 2009 [68] to support policy administration and dynamic delegation. The former controls the types of policies that individuals can create and modify, whereas the latter permits some users to create policies of limited duration to delegate selected capabilities to others. The delegation model used in [68] is a discretionary access control (DAC) model. Consequently, the profile only allows the owner of a permission to delegate it to a specific user, which is not scalable when permissions need to be delegated to a large number of users with the same job function. In many cases in which the delegator is not available, or is unable to perform the delegation, it is more convenient to have a third party, such as the administrator, initiating the delegation on behalf of the user.

This profile also lacks the support to allow delegators to delegate any subset of permissions assigned to him/her. On a separate issue, this profile does not have an enforcement mechanism. Enforcing administrative or delegation operations will update relevant policies which results in read-write conflicts while the access controller attempts to evaluate a user access request. Also when an administrator or delegator attempts to revoke a permission granted to a user, the same user might still be exercising the permission to access a resource, which violates system safety.

Another work related to XACML is [80], in which Xu et al. extend [68] to include the use cases of role-based delegation extending the delegation framework of [68], and policy administration with or without delegation extending the Use Cases proposed in [81]. Furthermore, they show how these extended Use Cases can be realized by extending the design implemented in [81] that retains the system safety by revoking permissions invalidated by policy updates. To provide the extra Use Cases and enforce delegation, we divide the access requests into three categories as follows:

Another administrative model has been proposed Crampton and Loizounor in [36]. This model, called SARBAC (Scoped Administrative RBAC), allows for more flexibility in administrative operations in that some of the operations that should be centrally managed in ARBAC97 (e.g., modification to the introduced relations) may be decentrally managed in SARBAC. Central to SARBAC is the concept of an administrative scope, which is defined using the role hierarchy, and is used for defining administrative domains. The administrative scope of a role r (denoted by $\sigma(r)$) consists of all roles that are descendants of r and are not descendants of any role that is incomparable with r. A role $r \in \sigma(a)$ if in the role hierarchy every path upwards from r goes through a. Each role is in the scope of the role itself. We say a scope is nontrivial if it includes more than one role. Using scopes for administration works best when the role hierarchy is a tree, with an all-powerful role at the root.

In [59] Li et al. present their approach for administering RBAC, called the UARBAC. UARBAC consists of a basic model and one extension: $UARBAC^P$, which adds parameterized objects and constraint-based administrative domains. UARBAC adopts the approach of administering RBAC with RBAC. By this, we mean that permissions about users and roles are administered in the same way as permissions about other kinds of objects. For example, the parameterized permission [business role, unit < Branch Hamburg, create] allows one to create a business role with the parameter unit having a value that is a descendant of Branch Hamburg.

These models are valid for generic domains, but due to the uniqueness of emergency management scenario, we decide to develop our administrative model based on emergency scopes.

2.7 Policy Similarity

One of the strategy used in our model to detect whether a denied access request represents an attempted abuse or an information need for an unspecified emergency is called policy based analysis. This analysis is based on measures to calculate how much a denied access request is close to satisfy existing policies. In literature of access control policy analysis, there are techniques for *policy similarity* analysis [61, 13, 60, 34] which are similar to our approach. Policy similarity is a process through which policies are compared with respect to the sets of requests they authorize. Given two policies, the process determines what is the relationship between them (e.g., equivalence, refinement, conflict, etc.). There are mainly two approaches to determine similarity between policies: based on policy similarity measures [61, 34] and based on Multi-Terminal Binary Decision Diagrams (MTBDD) based techniques [60]. We analyze the latter since it relies on similarity measures which are similar to our satisfaction level measures.

A first effort in the definition of a policy similarity measure has been proposed in [61] by Lin et al. Given two policies, this approach groups the same components in the two policies, and evaluates their similarity, then the obtained similarities are combined according to a weighted combination in order to produce an overall similarity score. This technique has been extended in [34] by Cho et all in order to perform policy similarity in a privacy-preserving manner allowing similarity evaluation of encrypted policies.

Policy similarity measures are different from our proposal since they aim to find relationships among policies and not satisfaction level of a denied access request w.r.t. existing policies. Although, the different purpose, they have aspects in common with our approach, i.e., a similar approach to measure the distance between two categorical or numerical values. Conversely, they are completely different regarding predicates since we need to measure how much a dar attribute value is close to satisfy a predicate and not the similarity between two Boolean expressions.

2.8 Anomaly Detection for Data Streams

One of the strategy used in our model to detect whether a denied access request represents an attempted abuse or an information need for an unspecified emergency is called anomaly based analysis. This analysis combines anomaly detection techniques with complex event processing in order to detect anomalous events which might represent emergency situations. According to [82] anomaly detection techniques for data streams can be categorized into nearest neighbor based, clustering based, spectral decomposition based and statistical based approaches.

Nearest neighbor based approaches [56, 3] are the most commonly used approaches, in this model, a data instance is declared as an anomaly if it is located far from its neighbors according to a distance function. Clustering based approaches [44, 77]s group similar data instances into clusters. Data instances are identified as anomalies if they do not belong to any cluster or their clusters are significantly smaller than other clusters. Nearest neighbor and clustering based techniques suffer from the choice of the appropriate input parameters, e.g., the appropriate distance function or cluster width.

Spectral decomposition based approaches [42, 54] is based on Principal Component Analysis (PCA) to reduce dimensionality before anomaly detection and finds a new subset of dimension which capture the behavior of the data. More precisely, the top few principal components capture the build of variability and any data instance that violates this structure for the smallest components is considered as an anomaly [32]. Spectral decomposition based techniques are computationally very expensive.

Statistical based approaches assume or estimate a statistical model which captures the distribution of the data. A data instance is declared as an anomaly if the probability of the data instance to be generated by the data model is low. Statistical based techniques are dived into parametric [70] and non-parametric [11] whether they assume or not the availability of the knowledge about the underlying data distribution. However, in many real life scenarios, no a priori knowledge of the sensor stream distribution is available, thus parametric approaches may be useless. Non-parametric techniques do not make any assumption about the underlying distribution and are computationally efficient.

Chapter 3

Emergency Access Control Model

In this chapter, the access control model to enforce flexible information sharing in support of emergency management is presented.

The key feature of this access control model is the detection of emergency situation based on Complex Event Processing (CEP) technology. How we use CEPs for emergency detection is explained in Section 3.1. This section introduces also the Core Event Specification Language (CESL), we have defined to specify emergency situations (see Subsection 3.1.1).

The core model and its extensions are described in Section 3.2. More precisely, this section presents formal definitions of emergency polices and an in depth analysis of the model (see Subsection 3.2.1). Moreover, extensions for administrative access control (Subsection 3.2.2) and policy composition (Subsection 3.2.3) are presented.

3.1 Emergency Detection

Emergency detection is major concern of our access control model. It can benefit from the recent advent of Complex Event Processing (CEP) systems [63] as they allow capturing complex event patterns signaling the beginning/ending of an emergency. CEP systems are able to continuously process flowing data from geographically distributed sources. CEP systems represent an evolution of Data Stream Management Systems (DSMSs) as explained in Section 2.1. The literature offers several languages for event pattern specification, but so far a standard event specification language has not yet emerged. To overcome the problem of lack of standardization, in the report a Core Event Specification Language (CESL) will be used [27].

3.1.1 Core Event Specification Language

Critical scenarios are often observed analyzing relevant information (e.g., body measures) collected by means of a set of sensor networks. In general, gathered data are sent to a monitor unit in the from of data streams (i.e., an append-only sequence of tuples with the same schema). This allows the monitor unit to continuously analyze them and immediately trigger the emergency when some particular constraints hold. Therefore, a CEP can play the role of monitor unit.

Example 3.1.1 *Let us consider as reference scenario patient remote monitoring. We assume that patients wear several monitoring devices that catch their health measures (e.g., temperature, heart rate, blood pressure, glucose, etc.). All gathered measures are encoded as tuples in a data stream and sent to a CEP, which can easily detect any anomaly signaling an emergency situation. As an example, an event modeling an emergency situation is given by heart rate measure lower than 60 beats per minute (bpm).*

According to the above scenario, conditions triggering the emergency are expressed as constraints on streams. Following stream terminology, we say that an *emergency event* happens when a tuple satisfying an emergency condition arrives. The emergency event can be simple, as a query on a single data stream (e.g., `s.heart_rate < 60`, where s is the patient vital signs stream and *heart_rate* one of its attributes), as well as more complex, like an aggregation query on a joint set of streams (e.g., an event to detect an epileptic attack should join many information about patient vital signs stream and movement sensors stream). To catch these emergency events, the proposed representation has to be able to model queries on streams. This can be done through stream query languages. However, in addition to conditions on streams, we are also interested in the detection of relationships between simple/complex events that might happen with different temporal order. For instance, let us assume that patients wear also movement sensors. To catch the emergency of the patient fall, it is necessary to detect this sequence of events: the patient falls to the ground and he/she does not stand up within the next 2 minutes. To catch also this kind of emergency, the proposed emergency description supports specification of event patterns.

CESL supports the following operators: (i) stream operators such as *selection*, *projection*, *aggregation*, and *join*; (ii) basic event operators, like *event type*, *event instance* and *array of event instances*; and (iii) complex event pattern operators, such as *sequence*, *negation* and *iteration* of events.

Stream Operators

Stream operators include typical SQL-like query operators such as *selection*, *projection*, *aggregation*, and *join*, but before introducing these operators in stream domain, we introduce some basic definitions of CESL (i.e., stream and predicate).

Definition 3.1.1 ***Stream:*** *a stream S with attributes* $Att(S) = \{A_1, ..., A_n\}$ *is a real-time, continuous, ordered (potentially unbounded) sequence of tuples.*

Example 3.1.2 *Consider Example 3.1.1, a possible stream catching patients health measures might be called* `VitalSigns` *and might contain the following attributes* `Att(VitalSigns) = {heart_rate, temperature, glucose_level, diastolic_pressure, systolic_pressure, respiratory_rate, patient_id}`.

Definition 3.1.2 ***Predicate:*** *is an expression* $P = a\ \theta\ b$ *where* a *is an attribute* $\in Att(S)$, $\theta \in \{< \mid > \mid = \mid \leq \mid \geq\}$, b *is an attribute* $\in Att(S)$ *or a constant value.*

Example 3.1.3 *Consider stream* `VitalSigns` *presented in Example 3.1.2, a possible predicate over this stream might check if the temperature of a patient is greater than a predefined threshold, i.e.,* `temperature > 37`.

CESL allows selecting relevant tuples from the history of all received ones according to a set of constraints contained in the body of an expression. The selection operator is formally defined as follows.

Definition 3.1.3 ***Selection Operator:*** *a selection* $\sigma(P)(S)$ *returns tuples belonging to stream* S *that satisfies predicate* P.

Example 3.1.4 *Consider stream* `VitalSigns` *presented in Example 3.1.2 and predicate* `temperature > 37`, *they might be combined using a selection operator in order to detect tuples with temperature value greater than 37 degrees, i.e.,* σ `(temperature > 37) (VitalSigns)`.

CESL supports also the projection operator which chooses a subset of the attributes in a tuple, and discards the rest. The projection operator is formally defined as follows.

Definition 3.1.4 ***Projection Operator:*** *a projection* $\pi(A_1, ..., A_n)(S)$ *returns attributes* $\{A_1, ..., A_n\} \in Att(S)$ *over stream* S.

Example 3.1.5 *Consider stream* `VitalSigns` *presented in Example 3.1.2, it might be necessary to choose only the temperature value and the patient_ id, i.e.,* π`(temperature, patient_id)(VitalSigns)`.

Before introducing *aggregation* and *join* operators in stream domain, we need to formally introduce the concept of window. CESL supports 4 types of windows:

- **Time-based window:** is simply expressed as $[s, o]$ where o (offset) is the number of time units[1] which represents how the window advances to from a new window, while s is the number of time units which denotes the window size. For example if the time unit is second, a window $[2s, 4s]$ means that a new window is created every 2 seconds and the window duration is 4 seconds.

- **Tuple-based window:** is expressed also as $[s, o]$ where o is the number of tuples which represents how the window advances to from a new window, while s is the number of tuples which defines the window size. For example a window $[3, 6]$ means that a new window is created every 3 tuples and the window size is 6 tuples.

- **Event-based window:** is specified as $[e_1, e_2]$ which means that the window is created when event e_1 occurs and is closed when event e_2 occurs.

- **Mixed window:** is specified as $[e_1, s]$ which means that the window is created when event e_1 occurs and the window size is s. The size s might be a number or a time expression whether window size is expressed in tuple-based or time-based mode.

Aggregation functions return a single value, calculated from values of a certain attribute in a specific window. CESL support typical SQL aggregate functions such as `sum, avg, count, max, min`. The aggregation operator has the following formal definition.

Definition 3.1.5 ***Aggregation Operator:*** *has the from* $\sum(F, A)(S)[w]$, *where the attribute A belonging to stream S is aggregated according to function F over window w.*

[1] *time_unit* $\in$ *{ms, s, mi, h, d, w, mo, y}*, ms = milliseconds, s = seconds, mi = minutes, h = hours, d = days, w = weeks, mo = months, y = years.

Example 3.1.6 *Consider stream VitalSigns presented in Example 3.1.2, it might be necessary to measure the average systolic pressure of a patient in the last hour, therefore the following aggregation operator might be used:* $\sum$*(avg, systolic_pressure) (VitalSigns) [1h, 1h].*

The join operator combines attributes of two streams into one. The formal definition of selection operator is the following.

Definition 3.1.6 ***Join Operator:*** *has the from* $Join(P)(S1[w_1], S2[w_2])$*, and it joins, with respect to predicate* P*, tuples belonging to window* w_1 *over stream* $S1$ *with tuples belonging to window* w_2 *over stream* $S2$*.*

Basic Event Operators

In the following, formal definitions of basic event operators such as *event type*, *event instance* and *array of event instances* and *window array* are provided.

Definition 3.1.7 ***Event Type:*** *an event type* ET *is the result of a query over one or more streams.*

Example 3.1.7 *Consider the result of query presented in Example 3.1.4, it might be assigned to the event type VS (VitalSigns) in the following way: VS =* σ*(temperature > 37) (VitalSigns).*

Definition 3.1.8 ***Event Instance:*** *an event instance* $ET\ e$ *is a tuple satisfying the query in an event type* ET*.*

Example 3.1.8 *Consider the event type VS presented in Example 3.1.7, each tuple belonging to VS might be assigned to an event instance vs in the following way: VS vs.*

Definition 3.1.9 ***Array of Event Instances:*** *an array of event instances* $ET\ e[\,]$ *contains the set of event instances of the event type* ET*.*

Example 3.1.9 *Consider again the event type VS presented in Example 3.1.7, the set of tuples belonging to VS might be assigned to an array of event instances vs[ ] in the following way: VS vs[ ].*

.

Definition 3.1.10 ***Window Array:*** *a window array* $ET\ e[\,][w]$ *specifies that array* $e[\,]$ *contains event instances of the event type* ET *that occur in window* w*.*

Example 3.1.10 *Consider again the event type* `VS` *presented in Example 3.1.7, the set of tuples belonging to* `VS` *occurred in a time window* `[1h,1h]` *might be assigned to an array of event instances* `vs[ ]` *in the following way:* `VS vs[ ][1h,1h].`

Event Pattern Operators

Event pattern operators are able to capture complex relations among events occurred in specific time windows. CESL supports event pattern operators, such as *sequence*, *negation* and *iteration*. Sequence operator allows capturing sequences of events using mixed windows as explained in the following formal definition.

Definition 3.1.11 ***Sequence:*** *has the from* $ET_1\ e_1,\ ET_2\ e_2[e_1, S_1], \ldots,$ $ET_n\ e_n[e_{n-1}, S_{n-1}]$*. A set of subsequent events matches with the sequence operator if event* e_2 *(of the event type* ET_2*) occurs within* S_1 *time units after occurrence of event* e_1*, event* e_3 *occurs within* S_2 *time units after* e_2 *and so on, defining in this way the sequence* $e_1, e_2, e_3, ..., e_n$*.*

Example 3.1.11 *Consider the healthcare scenario presented in Example 3.1.1 and the* `VitalSigns` *stream presented in Example 3.1.2. In order to detect whether or not the temperature of a patient is increasing in the last 10 minutes, the following sequence operator might be used.*

```
VS1 v1, VS2 v2[v1,5m], VS3 v3[v2,5m];
VS1 = σ(37.0 ≤ temperature ≤ 38.0)(VitalSigns);
VS2 = σ(39.0 ≤ temperature ≤ 40.0)(VitalSigns);
VS3 = σ(temperature ≥ 41.0)(VitalSigns);
```

The three event types `VS1, VS2, VS3` *contains respectively events with temperature between 37 and 38 degrees, between 39 and 40 degrees and greater than 41. The sequence* `VS1 v1, VS2 v2[v1,5m], VS3 v3[v2,5m]` *matches if an event instance* `v2` *with temperature between 39 and 40 degrees is received within 5 minutes after another event instance* `v1` *with temperature between 37 and 38 degrees and if an event* `v3` *with temperature greater than 41 is received within 5 minutes after* `v2` *defining in this way a sequence of events with increasing temperature.*

CESL allows representing the non-occurrence of an event in a given time interval through the negation operator whose formal definition is the following.

Definition 3.1.12 ***Negation:*** *a negation operator* $\neg\ ET\ e[w]$ *matches if event* e *has* ***not*** *occurred in window* $[w]$*.*

Example 3.1.12 *In a meteorological station a dry period is detected when it has not rain within one month since the last rain fell. This situation is expressed in CESL as follows.*

```
Rain r1, ¬ Rain r2 [r1,1mo]
Rain = σ(rain_level > 10)(RainSensors)
```

The event type `Rain` *matches when the attribute* `rain_level` *is greater than 10 millimeters, i.e., it is raining. This expression matches if an event of the event type* `Rain` *has* **not** *occurred within one month after* `r1`*, i.e., the last rain event.*

Definition 3.1.13 ***Iteration:*** *has the from ET* $e[\ ][w]\{P'\}$ *where* P' *is a predicate* $P' = \alpha\ \theta\ \beta$ *and* $e[\ ]$ *is an array of event instances of the event type ET that occur in window* w. $\alpha = (att, i)$, $\theta \in \{< \mid > \mid = \mid \leq \mid \geq\}$, $\beta = (att, j)$ *or a constant. att is an attribute* $\in Att(ET)$, *i and j are indexes* $\in I$, *such that:* $I = \{1, ..., |e|\} \cup \{*, ..i, i - x\}$, $x \in \{0, ..., i - 1\}$*), and* $j < i$. *An index i represents a specific event in* $e[\ ]$ *when* $i \in \{1, ..., |e|\}$, *all events in* $e[\ ]$ *when* $i = *$, *all events before* $e[i]$ *when* $i = ..i$, *or the event occurred x-events before* $e[i]$ *when* $i = i - x$.

Example 3.1.13 *An irregular heartbeat of a patient might be detected if the current heart rate is greater than the average heart rate of the patient in the last day. This situation is expressed in CESL as follows.*

```
HR1 hr1[1d,1d] {
  hr1[i].heart_rate > AVG(hr1[..i].heart_rate)
};
HR1 = π(heart_rate, VitalSigns);
```

The event type `HR1` *selects only the* `rain_level` *attribute from* `VitalSigns` *stream. If the* i^{th} *event, i.e., the current event* `hr1[i]`*, has a heart rate value greater than the average heart rate of the previous events* `hr1[..i]` *in a time window of one day, i.e.,* `[1d,1d]`*, then the heart rate value might be irregular.*

An overview of CESL operators is reported in Table 3.1.

3.1.2 Emergency Description

In light of our formal definitions of CESL operators, an emergency is modeled as a couple of events, defined in CESL that signal the beginning and ending of the emergency situation, respectively. More formally.

Stream Operators		
stream	S	a stream S with attributes $Att(S) = \{A_1, ..., A_n\}$ is a real-time, continuous, ordered (potentially unbounded) sequence of tuples.
predicate	$P = a\ \theta\ b$	a is an attribute $\in Att(S)$, $\theta \in \{<\mid>\mid=\mid\leq\mid\geq\}$, b is an attribute $\in Att(S)$ or a constant.
selection	$\sigma(P)(S)$	selection of stream S that satisfies predicate P.
projection	$\pi(A_1, ..., A_n)(S)$	projection of stream S over attributes $\{A_1, ..., A_n\} \in Att(S)$.
window w	time-based window $[s, o]$	s (size) is the window size, denoted as number of time units, o (offset) is the number of time units which represents how the window advances to from a new window.
	tuple-based window $[s, o]$	s is the window size, denoted as number of tuples, o is the number of tuples which represents how the window advances to from a new window.
	event-based window $[e_1, e_2]$	window is created when event e_1 occurs and is closed when event e_2 occurs.
	mixed window $[e_1, s]$	window is created when event e_1 occurs and the window size is s.
aggregation	$\sum(F, A)(S)[s, o]$	aggregation of attribute A of stream S according to function F over windows generated with size s and offset o.
join	$Join(P)(S1[s_1, o_1], S2[s_2, o_2])$,	join with respect to predicate P over tuples of sliding windows of stream $S1$(i.e., $S2$) generated with size $s1$(i.e., $s2$) and offset $o1$(i.e., $o2$).
Basic Event Operators		
event type	ET	an event type is a query over one or more streams.
event instance	$ET\ e$	an event instance is a tuple satisfying the query ET.
array	$ET\ e[\ ]$	declaration of the array $e[\ ]$ of event instances of the event type ET.
	$ET\ e[\ ][w]$	declaration of the array $e[\ ]$ of event instances of the event type ET that occur in window w.
Event Pattern Operators		
sequence	$ET_1\ e_1,\ ET_2\ e_2[e_1, S_1]$, $..., ET_n\ e_n[e_{n-1}, S_{n-1}]$	it matches if event e_2 (of the event type ET_2) occurs within S_1 time units after occurrence of event e_1, event e_3 occurs within S_2 time units after e_2 and so on, defining in this way the sequence $\{e_1, e_2, e_3, ..., e_n\}$.
negation	$\neg\ ET\ e[w]$	matches if event e has **not** occurred in window $[w]$.
iteration	$ET\ e[\ ][w]\{P'\}$ where P' is a predicate $P' = \alpha\ \theta\ \beta$ and $e[\ ]$ is an array of event instances of the event type ET that occur in window w.	$\alpha = (att, i)$, $\theta \in \{<\mid>\mid=\mid\leq\mid\geq\}$, $\beta = (att, j)$ or a constant. att is an attribute $\in Att(ET)$, i and j are indexes $\in I$, such that: $I = \{1, ..., \lvert e\rvert\} \cup \{*, ..i, i{-}x\}$, $x \in \{0, ..., i{-}1\}$), and $j < i$. An index i represents a specific event in $e[\ ]$ when $i \in \{1, ..., \lvert e\rvert\}$, all events in $e[\ ]$ when $i = *$, all events before $e[i]$ when $i = ..i$, or the event occurred x-events before $e[i]$ when $i = i - x$.

Table 3.1: CESL operators

Definition 3.1.14 ***(Emergency):*** *An emergency emg is a tuple (init, end,* timeout*,* identifier*), where init and end are emergency events specified in CESL, such that init denotes the event triggering the emergency and end is the optional event that turns off the emergency,* timeout *is the time within the emergency expires even though end has not occurred.*[2] *Identifier is an attribute belonging to both the schemes of the event type corresponding to init and end events.*

Every time an emergency event is triggered an emergency instance is created. Several instances of the same emergency event could hold at the same time, but with different values for attributes in *identifier*. The *identifier* component plays a key role in that it ensures to trigger an emergency instance only once and ensures also a connection between *init* and *end* events as the following example clarifies.

Example 3.1.14 *Let us consider again the patient monitoring scenario presented in Example 3.1.1 and the* `VitalSigns` *stream presented in Example 3.1.2. A bradycardia emergency can be defined as follows.*

```
BradycardiaEmergency {
  init: VS1 $v_1$;
  VS1 = $\sigma$(heart_rate < 60)(VitalSigns);
  end: VS2 $v_2$;
  VS2 = $\sigma$(heart_rate $\geq$ 60)(VitalSigns);
  timeout: $\infty$;
  identifier: patient_id;
}
```

The emergency starts when the heart rate of a patient is lower than 60 bpm and it ends when the heart rate of the same patient (i.e., with the same patient_id) returns greater than or equal to 60 bpm. When the `Bradycardia Emergency` *is detected for patient 1 the following emergency instance is created.*

```
BradycardiaEmergencyInstance1 {
  emg: BradycardiaEmergency;
  identifier: 1;
}
```

The `BradycardiaEmergencyInstance1` *is deleted when* `BradycardiaEmergency` *ends for patient 1. In ordered to better understand the*

[2] timeout is a temporal expression of the from [n *time_unit*], where $n \in \mathbb{N} \cup \{\infty\}$ and *time_unit* ∈ *{ms, s, mi, h, d, w, mo, y}*, ms = milliseconds, s = seconds, mi = minutes, h = hours, d = days, w = weeks, mo = months, y = years.

`identifier` role, consider the following tuples generated by monitoring patients a and b[3]*:* $(61, a), (60, a), (59, a), (58, a), (57, b)$. *When the third tuple arrives an instance for `BradycardiaEmergency` is created for patient a, but when the fourth tuple is received no new instance is created since the heart rate was already lower than 60 bpm for patient a. In contrast, it is correct to create a new instance when the fifth tuple is received because it comes from a different patient, i.e., patient b.*

3.2 Emergency Policy

Our model enforces controlled information sharing during emergencies through *temporary access control policies* (*tacps*). More precisely, since different instances of the same emergency might require different temporary access control policies, we associate with an emergency a *temporary access control policy template* that will be properly instantiated when an emergency is detected.

Definition 3.2.1 ***(Temporary Access Control Policy Template):*** *A* tacp *template is a tuple (sbj, obj, priv, ctx, obl), with the following semantics: when the Boolean expression ctx defined on context is true, users identified by the subject specification sbj are authorized to exercise the privilege priv on the resource identified by object specification obj. In case the obligation obl is not null, it denotes a set of actions that must be fulfilled every time an authorized user exercises priv on the objects denoted by obj.*

Our proposal adopts a model similar to attribute-centric RBAC-A [57]. The model in [57] is a combination of role-based access control and attribute-based access control. We choose this model, because we need to identify users by their roles (e.g., doctor, patient, etc.) as well as specify attribute-based conditions. Therefore, in our proposal a subject specification *sbj* is a pair (*roles*, *cond*), where the first is a set of roles and the second is a condition related to the user profile attributes. An object specification *obj* is a pair (*object*, *cond*), where *object* denotes a target object and *cond* is a condition related to the object attributes.[4] The context is modeled as a set C of pairs *(att, val)*, where *att* is a context attribute (e.g., time, location, session information etc.) and *val* is the corresponding value.

[3] For briefity, we consider only the two attributes heart_rate and patient_id.

[4] *Cond* is a Boolean combination of predicates in the from $\alpha\ \theta\ \beta$, where α is an attribute belonging to the user profile (object, respectively), θ is a matching operator in $\{<, >, =, \leq, \geq\}$, whereas β is a constant value or an attribute *att*.

Example 3.2.1 *Consider the bradycardia emergency presented in Example 3.1.14. Suppose that, during this emergency, access to the Electronic Medical Record (EMR) of a patient (object condition) should be extended to the subjects taking care of him/her (subject condition - e.g., paramedics). Moreover, when a subject not normally authorized accesses the EMR, the corresponding patient should be warned with an email (obligation). In order to enforce these requirements, the following tacp template can be defined:*

```
BradycardiaPolicy {
  sbj: (paramedic, param_id = call.param_id);
  obj: (EMR, patient_id = emg.patient_id);
  priv: read;
  ctx: -;
  obl: mailto(patient_mail);
}
```

The tacp subject is the paramedic who answered to the emergency call, whereas the tacp object is the EMR of the patient under emergency condition. The context condition ctx is empty, whereas the obligation ensures that when a paramedic reads the patient EMR, an email is sent to the patient mail address. When the BradycardiaEmergency *is detected for patient 1, the following tacp instance is created and inserted into the policy base.*

```
BradycardiaPolicyInstance1 {
  sbj: (paramedic, param_id = 3);
  obj: (EMR, patient_id = 1);
  priv: read;
  ctx: -;
  obl: mailto(patient1@hospital.com);
}
```

This is created assuming 3 is the identifier of the paramedic on the ambulance answering the patient call and patient1@hospital.com is the email address of the patient under emergency.

We also bind emergencies with one or more *emergency obligations* in order to let the system immediately and automatically execute urgent activities required by emergency response plans. The binding of an emergency with the corresponding tacps and obligations is modeled by the so-called emergency policies.

Definition 3.2.2 ***(Emergency Policy):*** *An emergency policy is a tuple (emg, tacp, obl), where emg is an emergency description (cf. Definition 3.1.14), tacp can be one or more temporary access control policy templates* *(cf. Definition 3.2.1) and obl is an optional field which contains one or more emergency obligations, i.e., actions that must be performed when the emergency emg is detected.*

Example 3.2.2 *Based on Examples 3.1.14 and 3.2.1, an emergency policy might be the following.*

```
BradycardiaEP {
  emg: BradycardiaEmergency;
  tacp: BradycardiaPolicy;
  obl: call_ambulance(patient_address);
}
```

where BradycardiaPolicy *and* call_ambulance *are, respectively, the tacp template and the obligation to be enforced when* BradycardiaEmergency *is detected. When the* Bradycardia *emergency is detected, the following emergency policy instance is created and inserted into the policy base.*

```
BradycardiaEPInstance1 {
  emg: BradycardiaEmergencyInstance1;
  tacp: BradycardiaPolicyInstance1;
  obl: call_ambulance(40 Storrow Dr);
}
```

This is created assuming 40 Storrow Dr is the address of the patient under emergency.

3.2.1 Emergency Policy Correctness

The main function of emergency policies is the enforcement of the corresponding tacps/obligations upon emergency detection. More precisely, emergency policy enforcement consists of two main steps: (1) the creation/deletion of the corresponding emergency instances and (2) the consequent creation/deletion of instances of the corresponding temporary access control policies. Since emergency activation/deactivation is a time consuming operation, a particular attention has to be paid in properly defining the *init* and *end* emergency events to ensure that, even if syntactically well-defined, they will not imply a simultaneous activation and deactivation of an emergency. In general, this type of error occurs when the two sets of tuples satisfying *init* and *end* events are not disjoint. Indeed, in this case the arrival of just one tuple may cause the simultaneous creation and deletion of the corresponding emergency and tacp instances. Let us consider, as an example, an emergency specification where *init*: $temp \geq 37$ and *end*: $temp \leq 39$. In this case, the arrival of a tuple t such that $t.temp = 38$ results in the simultaneous creation and deletion of the corresponding emergency and *tacp* instances.[5] We formally define this problem in the following subsection.

Definition 3.2.3 ***(Simultaneous Holding Problem (SHP)):*** *Let e be a CESL event, we denote with VS_e the* validity set *of e, defined as the set of tuples satisfying the event e. Let emg be an emergency, and let VS_i, VS_e the validity sets corresponding to emg.init and emg.end, respectively. A Simultaneous Holding Problem (SHP) occurs when, at a certain time instant X, two tuples t_i and t_e, such that $t_i \in VS_i$ and $t_e \in VS_e$, arrive.*

In order to verify if the *init* and *end* events are not correctly defined, we propose a set of *Validity Checks*. We call the first one *pre-processing validity check*, since it is used to detect, before the emergency policy registration, if its emergency description might bring to a potential SHP, so as to prevent its registration in the system. To better clarify, let us consider once again the previous example. Here, it is possible to easily detect that there exist some tuples satisfying simultaneously both init and end events, as their validity sets are not disjoint. In general performing this check implies to compute the validity sets of init/end events and verify that they have no common tuples.

Pre-Processing Validity Check

We first analyze *pre-processing validity check* for simple events based on queries, then for complex event patterns. Simple events are based on queries such as selection and projection. Indeed, this process is not possible for aggregation and join operators because it is impossible to predict a priori the result of the aggregation function or which values will satisfy a join predicate. Thus, it is impossible to statically compute their validity sets.
As first step, the process computes the validity set for each attribute in $Atts(init) \cup Atts(end)$. A validity set is calculated through function $VS(a, ev)$ which returns the validity set of attribute a for the ev event. This is done is a different way depending on ev. Let us first consider events with a unique operator, i.e., selection or projection. In case ev contains a projection operator, $VS(a, \pi(a, a_1, \ldots, a_n)(S)) = Dom(a)$ where $Dom(a)$ denotes interval of attribute a (i.e., $Dom(a) = [min, max]$)[6] and S is the stream over which the event is defined. In case ev contains a selection operator, then $VS(a, \sigma(C)(S))$ depends on the clause C. If C is not defined over attribute a, then VS returns the whole domain $Dom(a)$. In case C is a simple predicate $a\ \theta\ c$, where

[5] Here and in the following we use dot-notation to indicate fields of events, emergencies or policies.

[6] If a is a non-numeric attribute, i.e., string, then the lexicographical order is used.

$\theta \in \{< \mid > \mid = \mid \leq \mid \geq\}$ and c is a constant value, then function *VS* is calculated in the following way:[7]

$$VS(a, \sigma(a\ \theta\ c)(s)) = \begin{cases} [min, c), & \text{if } \theta \text{ is } < \\ [min, c], & \text{if } \theta \text{ is } \leq \\ [c, c], & \text{if } \theta \text{ is } = \\ [min, c) \cup (c, max], & \text{if } \theta \text{ is } \neq \\ (c, max], & \text{if } \theta \text{ is } > \\ [c, max], & \text{if } \theta \text{ is } \geq \end{cases}$$

In case the clause C is a conjunction $p_1 \wedge \cdots \wedge p_n$, then function *VS* is calculated as $VS(a, \sigma(p_1)(S)) \cap \cdots \cap VS(a, \sigma(p_n)(S))$. In case the clause C is a disjunction $p_1 \vee \cdots \vee p_n$, then function *VS* is calculated as $VS(a, \sigma(p_1)(S)) \cup \cdots \cup VS(a, \sigma(p_n)(S))$. The more complex cases are straightforwardly calculated as combinations of intersections and unions. In case the *ev* is a combinations of selection or projection operators, then $VS(a, op_1 \oplus \ldots \oplus op_n)$ is calculated for each operator op_i and the results are, then, intersected as $VS(a, op_1) \cap \ldots \cap VS(a, op_n)$.

Example 3.2.3 *An high stress situation may be detected when a patient heart rate is greater than 90 beats per minute and the respiratory rate is greater than 20 breaths per minute, or when the frequency measured by the electroencephalogram (eeg) is lower than 60Hz. The stress situation ends when the values return in a normal range.*

```
StressEmergency {
    init: VS1 $v_1$
    VS1 = $\sigma((hr > 90 \wedge rr > 20) \vee (eeg < 60))$ (VitalSings);
    end: VS2 $v_2$
    VS2 = $\sigma((hr \leq 90 \wedge rr \leq 20) \vee (eeg \geq 60))$ (VitalSigns);
    timeout: $\infty$;
    identifier: patient_id;
}
```

Assuming $Dom(hr) = [0, 200]$, *Dom*$(rr) = [0, 100]$ *and Dom*$(eeg) = [0, 500]$, *then the validity sets extracted from* init *and* end *events are the following:*

$$\begin{array}{ll} VS^{hr}_{init} = [91, 200] & VS^{hr}_{end} = [0, 90] \\ VS^{rr}_{init} = [21, 100] & VS^{rr}_{end} = [0, 20] \\ VS^{eeg}_{init} = [0, 59] & VS^{eeg}_{end} = [60, 500] \end{array}$$

[7] If c is not a constant value, i.e., it is an attribute, then it is not possible to calculate the validity set, thus another validity check, i.e., event rewriting or post processing, is executed.

The intersections calculated for each validity set are the following:

$$\begin{aligned} VS^{hr}_{init} \cap VS^{hr}_{end} &= [91, 200] \cap [0, 90] = \emptyset \\ VS^{rr}_{init} \cap VS^{rr}_{end} &= [21, 100] \cap [0, 20] = \emptyset \\ VS^{eeg}_{init} \cap VS^{eeg}_{end} &= [0, 59] \cap [60, 500] = \emptyset \end{aligned}$$

Since the intersections between the attributes validity sets are empty, the two events cannot simultaneously hold.

The *pre-processing validity check* can be performed also for event patterns such as sequence, negation, iteration. In the following we consider each possible case.

Sequence

We consider the case where *init* and *end* are defined as sequences $Ie_1, Ie_2, \ldots,$ Ie_n and $Ee_1, Ee_2, \ldots, Ee_m$, respectively. In this case, the *pre-processing validity check* can be executed only if the two sequences have the same length (i.e., $n = m$) and any couples of events in the same position in the two sequences (i.e., Ie_i, Ee_j, $i = j$) contain only projection and/or selection operators defined over the same window.[8] In case the two sequences have a different length or events in the same position contain aggregation and/or join operators and they are defined over different windows, another validity check, i.e., event rewriting or post processing, is executed. The *pre-processing validity check* for sequences is performed as follows: every event Ie_i in the *init* sequence is compared to the corresponding (i.e., in the same position) event Ee_j in the *end* sequence. If for each couple of events (Ie_i, Ee_j), they are defined over the same window and they might simultaneously hold (i.e., for each attribute $a \in Atts(Ie_i) \cup Atts(Ee_j)$, the intersection $VS^a_{Ie_i} \cap VS^a_{Ee_j} \neq \emptyset$), then *init* and *end* events might bring to a potential SHP.

Example 3.2.4 *Consider an emergency detected when a patient temperature increases, i.e., a sequence of events with increasing temperature values is received.*

```
IncreasingTemperature {
    init: (VS1 v₁, VS2 v₂[v₁,5m], VS3 v₃[v₂,5m],
              _, patient_id);
    VS1 = σ( 35 ≤ temp ≤ 37)(S);
    VS2 = σ( 38 ≤ temp ≤ 40)(S);
```

[8] if Ie_i and Ee_j are defined over different windows, it is not possible to receive tuples which satisfy both Ie_i and Ee_j, therefore the SHP cannot occur.

```
    VS3 = $\sigma$( temp $\geq$ 41)(S);
    end: (VS4 $v_4$, VS5 $v_5$[$v_1$,5m], VS6 $v_6$[$v_2$,5m],
            _, patient_id);
    VS4 = $\sigma$( 34 $\leq$ temp $\leq$ 37)(S);
    VS5 = $\sigma$( 38 $\leq$ temp $\leq$ 41)(S);
    VS6 = $\sigma$( temp $\geq$ 42)(S);
    timeout: $\infty$;
    identifier: patient_id;
}
```

In this example, the arrival, for a certain patient, of a sequence of tuples with temperatures $36, 39, 44$ *results in the SHP. Since the two sequences have the same length, they contain only selection operators and events in the same position are defined over the same window, it is possible to perform the* pre-processing validity check. *In this example, the validity check fails, because: (1)* $VS1$ *and* $VS4$ *are in the same position, they are defined over the same window and they can simultaneously hold (i.e.,* $VS^{temp}_{VS1} \cap VS^{temp}_{VS4} = [35, 37] \neq \emptyset$*); (2)* $VS2$ *and* $VS5$ *are in the same position, they are defined over the same window (i.e., [*v_1*,5m]) and they can simultaneously hold (i.e.,* $VS^{temp}_{VS2} \cap VS^{temp}_{VS5} = [38, 40] \neq \emptyset$*), (3)* $VS3$ *and* $VS6$ *are in the same position, they are defined over the same window (i.e., [*v_2*,5m]) and they can simultaneously hold (i.e.,* $VS^{temp}_{VS3} \cap VS^{temp}_{VS6} = [42, 100] \neq \emptyset$*).*

Negation

In this case, *init* and *end* operators have the from $\neg\ ET_1\ e_1[w_1]$ and $\neg\ ET_2\ e_2[w_2]$. The *pre-processing validity check* can be executed only if the two negations contain only projection and/or selection operators and they are defined over the same window. Otherwise, *event rewriting* or *post-processing* is executed. The *pre-processing validity check* for negations is performed as follows: if e_1 and e_2 are defined over the same window (i.e., $w_1 = w_2$) and e_1 and e_2 might simultaneously hold (i.e., for each attribute $a \in Atts(e_1) \cup Atts(e_2)$, the intersection $VS^a_{e_1} \cap VS^a_{e_2} \neq \emptyset$), then they may cause a SHP.

Example 3.2.5 *Consider the following generic emergency based on a wrong definition of two negation event patterns.*

```
GenericEmergency {
    init: ($\neg$ $ET_1$ $e[w]$, _, patient_id);
    $ET_1$ = $\sigma(x > 10)$
    end: ($\neg$ $ET_2$ $e[w]$, _, patient_id, _,
            patient_id);
    $ET_2$ = $\sigma(x > 20)$
    timeout: $\infty$;
}
```

If during the window $[w]$ *a tuple with the x attribute (*Dom(x)*=[0,100]) greater than 20 is not received, then the SHP occurs. In this case, since the two negations are defined over the same window and they contain only selection operators, it is possible to perform the* pre-processing validity check. *Here, the validity check fails, because* ET_1 *and* ET_2 *are defined over the same window (i.e.,* $[w]$*) and they can simultaneously hold (i.e.,* $VS^x_{ET_1} \cap VS^x_{ET_2} = [21, 100] \neq \emptyset$*).*

Iteration

In this case, *init* and *end* have the from ET_1 $e_1[w_1]\{P_1\}$ and ET_2 $e_2[w_2]\{P_2\}$. The *pre-processing validity check* can be executed only if the two iteration predicates P_1 and P_2 and the two events ET_1 and ET_2 contain only projection and/or selection operators. Moreover, the two iterations should be defined over the same window. Otherwise, *event rewriting* or *post-processing* is executed. The *pre-processing validity check* for iterations checks the following conditions: (1) e_1 and e_2 are defined over the same window (i.e., $w_1 = w_2$) and they might simultaneously hold (i.e., for each attribute $a \in Atts(e_1) \cup Atts(e_2)$, the intersection $VS^a_{e_1} \cap VS^a_{e_2} \neq \emptyset$); (2) iteration predicates P_1 and P_2 might simultaneously hold (i.e., for each attribute $a \in Atts(P_1) \cup Atts(P_2)$, the intersection $VS^a_{P_1} \cap VS^a_{P_2} \neq \emptyset$). If both the conditions hold, then they might bring to a potential SHP.

Example 3.2.6 *Consider an emergency of tachycardia which starts when a patient heart rate is greater than or equal to 90 bpm, for one minute, and it ends when the same patient heart rate is lower than or equal to 100 bpm, for one minute. This is a wrong definition of Tachycardia Emergency.*

```
Tachycardia {
    init: (VitalSigns e[ ][1m,1m]{e[i].hr ≥ 90}, _,
           patient_id);
    end:  (VitalSigns e[ ][1m,1m]{e[i].hr ≤ 100}, _,
           patient_id);
    timeout: ∞;
}
```

In this example, the domain of the heart rate attribute is [0,140] *and if during the window* $[1m, 1m]$ *every received tuple has attribute* $hr = 95$*, then SHP occurs. In this case, since the two iterations are defined over the same window and they contain only selection operators, it is possible to perform the* pre-processing validity check. *In this example, the validity check fails, because* ET_1 *and* ET_2 *are defined over the same window (i.e.,* $[w]$*) and iteration predicates can simultaneously hold (i.e.,* $VS^{hr}_{P_1} \cap VS^{hr}_{P_2} = [90, 100] \neq \emptyset$*).*

Event Rewriting Validity Check

Unfortunately, the *pre-processing validity check* is not possible for those events whose validity sets cannot be computed *"a priori"*. For instance, this is the case of an emergency monitoring a blood pressure disease with *init*: $systolic \geq diastolic + 50$ and *end*: $diastolic \geq systolic - 50$. To handle these cases, we propose two additional strategies to detect the SHP. The idea is to detect tuples satisfying both init and end events at run time, so as to discard them for emergency activation/detection purpose. More precisely, the first approach, called *event rewriting*, implies to rewrite *init* and *end* events in such a way that they are not triggered by the arrival of these tuples. This is achieved by transforming *init* in $init \wedge \neg end$ and *end* in $end \wedge \neg init$, as the following example clarifies.

Example 3.2.7 *Consider the wrong emergency specification presented above, where the init and end events are defined in CESL as: init* $= \sigma(systolic \geq diastolic + 50)(S)$ *and end* $= \sigma(system \leq diastolic + 50)(S)$ *where* S *is the stream over the two events are defined. According to the strategy above explained, they are rewritten as follows:*

$$\begin{aligned}&new\_init = \sigma(systolic \geq diastolic + 50)(S) \wedge \\ &\neg\sigma(diastolic \geq systolic - 50)(S) \\ &new\_end = \sigma(diastolic \geq systolic - 50)(S) \wedge \\ &\neg\sigma(systolic \geq diastolic + 50)(S)\end{aligned}$$

The arrival of a tuple t *with attributes systolic* $= 100$ *and diastolic* $=$ 50 *does not match neither with new_init nor with new_end, therefore the arrival of* t *does not result in the SHP.*

Post Processing Validity Check

However, this rewriting is not always possible, since it works only when *init* and *end* are defined over the same set of streams. Indeed, using CESL operators, it is not possible to combine with negation events from different streams. For this reason, we propose a further validity check, called *post-processing validity check*. This is enforced outside the CEP system since it requires checking tuples flowing out from *init* and *end* events to discard those that trigger a SHP.

Example 3.2.8 *Suppose that the monitoring system is used to monitor patients glucose level and insulin drips level, so as to raise a hypoglycemia emergency when the patient glucose level is lower than or equal to 70 mg/dl. When the emergency is detected the insulin drips level is automatically increased by* *an emergency obligation. When the insulin level returns greater than or equal to 1.2, then the emergency ends.*

```
HypoglycemiaEmergency {
    init: VS1 v_1;
    VS1 = σ(glucose_level ≤ 70)(VitalSigns);
    end: ID1 id_1;
    ID2 = σ(insulin_level > 1.2)(InsulinDrip);
    timeout: ∞;
    identifier: patient_id;
}
```

In this case, the simultaneous arrival of two tuples t_1 from stream VitalSigns *and t_2 from stream* InsulinDrip*, with the same* patient_id *and attributes t_1.*glucose_level *= 68 and t_2.*insulin_level *= 1.3 results in the SHP.*

To handle this type of SHP, we introduce a module called *Post Processing (PP)* that detects tuples causing SHP and, based on setting introduced by the emergency manager, will always perform one of the following actions: (i) discard tuples causing the SHP, i.e., the emergency is not activated, or (ii) discard only the tuple causing the end of the emergency, i.e., the emergency is activated and never deactivated. In both cases, a warning is sent to the emergency manager who has defined the policy causing the SHP. The solution (i) is secure because it prevents the activation of a tacp due to a wrong emergency definition and is efficient because it avoids time consuming operation such as emergency activation/deactivation. However, in case of a real emergency, it might endanger patients health. The solution (ii) does not endanger patient lives, but it is less secure due to the activation of a tacp which will never be deactivated. The choice between (i) and (ii) depends on the domain, for instance in healthcare domain action (ii) is better to not endanger patient lives, whereas in a military domain action (i) is better to not disclose confidential information for longer than necessary. The *post-processing validity check* is important for the overall system correctness, independently from the response strategy adopted, because it signals to emergency managers those emergencies causing SHP contributing to the proper definition of emergency descriptions.

Correctness Enforcement

Based on the above-described validity checks the overall process of emergency policy correctness validation is given in Algorithm 1. This is executed each time a user defines/modifies an emergency description, i.e., its *init* and *end* events.

Algorithm 1: SHP

```
   Input  : init and end events
   Output: {Valid, Invalid, Post}
 1 switch init, end do
 2     case init is simple event $\wedge$ end is simple event
 3         res = ShpSimple(init, end);
 4     case init is event pattern $\wedge$ end is event pattern
 5         res = ShpPattern(init, end);
 6     otherwise res = Post;
 7 endsw
 8 if res = Post then Post(init, end);
 9 return res;
```

Algorithm 1 first checks if both *init* and *end* events are simple events (line 2) or event patterns (line 4). Indeed, in these cases we can perform the *pre-processing validity checks* or the *event-rewriting* strategy. These are carried out by two distinct functions namely *ShpSimple* (line 3) and *ShpPattern* (line 5), otherwise the post-processing strategy is enforced (line 6). These functions return *Invalid* or *Valid*, whether *init* and *end* event can or cannot generate an SHP. In case it is not possible to determine *Valid*/*Invalid* status, these functions return *Post*. In these cases, the *post-processing validity check* is carried out by the *Post* procedure (line 8), which register *init* and *end* into a list of events whose outputs have to be monitored by the *Post Processing (PP)* module.

Function ShpSimple(init, end)

```
if GetStream (init) $\neq$ GetStream (end) then return Post;
if Atts (init) $\cap$ Atts (end) $= \emptyset$ then
    Rewrite (init, init $\wedge \neg$ end);
    Rewrite (end, end $\wedge \neg$ init);
    return Valid;
end
if GetOper (init) $\nsubseteq \{\sigma, \pi\}$ $\vee$ GetOper (end) $\nsubseteq \{\sigma, \pi\}$ then
    return Post;
foreach $p_i \in$ GetPred (init) $\cup$ GetPred (end) do
    if $p_i.\beta$ is not a constant then return Post;
end
foreach $a_i \in$ Atts (init) $\cup$ Atts (init) do
    if VS($a_i$, init) $\cap$ VS($a_i$, end) $\neq \emptyset$ then return Invalid;
end
return Valid;
```

As first check, the *ShpSimple* function verifies whether *init* and *end* events are defined over different streams. This is done by means of function *GetStream(e)*, which returns the stream over which event e is defined. If they are defined over different streams, a post processing validity check is needed (line 1). Then, the function verifies if *init* and *end* events are defined over different attributes (line 2). In this case, the *pre-processing validity check* cannot be performed. However, since events are defined over the same stream, we can implement the event-rewriting strategy. This is done by function *Rewrite* (lines 3, 4). If *init* and *end* events have at least one common attribute, then the operators used in *init* and *end* are analyzed to perform the *pre-processing validity check*. If they use operators which are not selection or projection, then a post processing validity check is needed (line 8) since the validity sets cannot be computed. Otherwise, each predicate p in both *init* and *end* is analyzed (lines 9-11). If at least one predicate is not a comparison between an attribute and a constant value, then a post processing validity check is needed (line 10). Otherwise, each attribute a over which a predicate in both *init* and *end* is defined, is analyzed (lines 12-14), so as to compute its validity set. For each attribute a, the intersection between the validity sets of *init* and *end* is calculated. If the intersection is not empty, then the two events might cause an SHP (line 13), so the invalid message is returned. In case all these checks fail, then *ShpSimple* function returns *Valid* (line 15).

Function ShpPattern(init, end)

```
1  switch init, end do
2      case init is sequence $\wedge$ end is sequence
3          if length (init) $\neq$ length (end) then return Post;
4          for $i = 1$; $i <$ length (init); $i{+}{+}$ do
5              if $init_i.w \neq end_i.w$ then return Post;
6              res = ShpSimple ($init_i$, $end_i$);
7              if $res \neq Valid$ then return res;
8          end
9          return Valid;
10     case init is negation $\wedge$ end is negation
11         if $init.w \neq end.w$ then return Post;
12         res = ShpSimple (init, end);
13         return res;
14     case init is iteration $\wedge$ end is iteration
15         if $init.w \neq end.w$ then return Post;
16         res1 = ShpSimple (init, end);
17         res2 = ShpPredicate (init.P, end.P);
18         if $res1 = Valid \wedge res2 = Valid$ then return Valid;
19         if $res1 = Invalid \vee res2 = Invalid$ then return Invalid;
20         else return Post;
21     otherwise return Post;
22 endsw
```

Function *ShpPattern* performs a process similar to the one of *ShpSimple*, but tailored to event patterns. First, it checks if *init* and *end* events are both defined as the same type of pattern: two sequences (line 2), two negations (line 10) or two iterations (line 14). In case the two events are defined over different types of patterns, then a post processing validity check is needed (line 21).

When *init* and *end* events are both defined as sequences (line 2), *ShpPattern* checks if the two sequences do not have the same length. In this case, a post processing validity check is needed (line 3). Otherwise, every event $init_i$ in the *init* sequence is compared against the corresponding end_i event in the *end* sequence (lines 4-8). If two events $init_i$ and end_i are not defined over the same window, then a post processing validity check is needed (line 5). Otherwise, the two simple events $init_i$ and end_i are analyzed through function *ShpSimple* (line 6). If *ShpSimple* returns *Invalid* or *Post*, then this result is returned by *ShpPattern* (line 7). In case all events in the *init* and *end* sequences are defined over the same windows and *ShpSimple* has always returned *Valid*, then *ShpPattern* returns *Valid* (line 9).

When *init* and *end* events are defined as negations (line 10), *ShpPattern* checks if the two events are not defined over the same window, so as to return a *Post* message (line 11). Otherwise, the two events are analyzed through *ShpSimple* and the result is returned by *ShpPattern* (lines 12, 13).

When *init* and *end* are defined as iterations (line 14), it verifies if the two events are not defined over the same window and, if this is the case, a post processing validity check is needed (line 15). Otherwise, the two events and the two iteration predicates are analyzed through *ShpSimple* and *ShpPredicate*[9] and the results are stored respectively in *res1* and *res2* variables (lines 16, 17). If both *res1* and *res2* are set to *Valid*, then *ShpPattern* returns *Valid* (line 18). If *res1* or *res2* are set to *Invalid*, then *ShpPattern* returns *Invalid* (line 19). Otherwise a post-processing validity check is needed (line 20).

Correctness Enforcement Formal Demonstrations

The following theorem proves the correctness of Algorithm 1.

Theorem 3.2.1 ***(Correctness of Algorithm 1).*** *Let e be an emergency description,* init *and* end *its events analyzed by Algorithm 1. For each tuple t that simultaneously triggers both* init *and* end *events, no instance of emergency e is created and simultaneously deleted.*

Before proving it we need to introduce two lemmas stating the correctness of functions *ShpSimple* and *ShpPattern*.

[9] Function *ShpPredicate* takes as input two predicates and returns *Valid* or *Invalid*, depending on whether their validity sets are disjoint or not.

Lemma 3.2.1 ***(ShpSimple Correctness).*** *Let e be an emergency description,* init *and* end *its simple events analyzed by* ShpSimple*. For each tuple t that simultaneously triggers both* init *and* end *events, no instance of emergency e is created and simultaneously deleted.*

Lemma 3.2.2 ***(ShpPattern Correctness).*** *Let e be an emergency description,* init *and* end *its event patterns analyzed by function* ShpPattern*. For each tuple t that simultaneously triggers both* init *and* end *events, no instance of emergency e is created and simultaneously deleted.*

Proof of Lemma 3.2.1 (ShpSimple Correctness)
We start the proof by recalling that if *ShpSimple* returns *Invalid* for a couple of *init* and *end* events, then also Algorithm 1 returns *Invalid* (see line 3). This implies that *init* and *end* will not be registered in the CEP, then no instance of emergency *e* is created and simultaneously deleted. If *ShpSimple* returns *post*, then Algorithm 1 executes function *Post* (line 8). This function ensures that the SHP cannot occur. Function *ShpSimple* returns *Valid* (line 20) in case of event rewriting or when the original *init* and *end* events cannot cause the SHP. Let us consider both the cases:

Event Rewriting: it is performed when *GetStream(init)* = *GetStream(end)* (line 1) and *Atts(init)* ∩ *Atts(end)* = ∅ (line 2). Since *GetStream(init)* = *GetStream(end)*, the SHP might occur when a tuple *t* satisfying *init* and *end* is received. This can be formalized by function *sat(t, e)* which returns true if tuple *t* satisfies event *e*. More precisely, in case of SHP we have that $sat(t, init) = true \wedge sat(t, end) = true$. After event rewriting, this equation will never hold for *newInit* and *newEnd*. Indeed, by construction *newInit* = *init* ∧ ¬ *end* (line 3) and *newEnd* = *end* ∧ ¬ *init* (line 4). Thus, $sat(t, newInit) = sat(t, init) \wedge \neg\ sat(t, end)$ and $sat(t, newEnd) = sat(end, t) \wedge \neg\ sat(init, t)$. Thus, if a tuple *t* that satisfies both *init* and *end* events arrives, it will never satisfy both *newInit* and *newEnd*, since this is given by $sat(t, init) \wedge\ \neg\ sat(t, end)\ \wedge\ sat(end, t) \wedge \neg\ sat(init, t)$ $= true \wedge false \wedge true \wedge false = false$.

Init and End are Valid: *ShpSimple* returns *Valid* when *GetStream(init)* = *GetStream(end)* (line 1), *Atts(init)* ∩ *Atts(end)* ≠ ∅ (line 2), *GetOperators(init)* ⊆ {σ, π} ∨ *GetOperators(end)* ⊆ {σ, π} (line 7), ∄ p_i ∈ {*GetPred(init)* ∪ *GetPred(end)* } such that $p_i.\beta$ is not a constant (lines 9-11) and ∀ a_i ∈ {*Atts(init)* ∪ *Atts(end)*}, $VS(a_i,\ init) \cap VS(a_i,\ end) = \emptyset$ (lines 12-14). If all the listed conditions are met, then ∀ a_i the *init* and *end* validity sets are disjoint, indeed for any attribute a_i belonging to the schema of the tuple

t or the attribute value belongs to $VS(a_i, init)$ or it belongs to $VS(a_i, end)$, as such a SHP will never occur.

Proof of Lemma 3.2.2 (ShpPattern Correctness)
We start the proof by recalling that if *ShpPattern* returns *Invalid* for a couple of *init* and *end* events, then also Algorithm 1 returns *Invalid* (see line 5). This implies that *init* and *end* will not be registered in the CEP, then no instance of emergency e is created and simultaneously deleted. If *ShpPattern* returns post, then Algorithm 1 executes the *Post* function (line 8). The *ShpPattern* function returns *Valid* (lines 9, 13, 18) in the following cases:

Sequence Event Patterns: when *init* and *end* events are sequences, the function returns *Valid*, if *length(init)* = *length(end)* (line 3), $\forall$ $init_i \in init$ and $\forall$ $end_i \in end$, $init_i.w = end_i.w$ (line 5) and $ShpSimple(init_i, end_i) = Valid$ (line 6). The Lemma 3.2.1 ensures that if $ShpSimple(init_i, end_i)$ returns valid, then $init_i$ and end_i cannot cause the SHP. If this is true for all the events in the same position in *init* and *end* sequences and they are defined over the same windows, then the SHP cannot occur for *init* and *end* events.

Negation Event Patterns: when *init* and *end* events are negations, the function returns *Valid* if $init.w = end.w$ (line 11) and *ShpSimple(init, end)* = *Valid* (line 12). The Lemma 3.2.1 ensures that if *ShpSimple(init, end)* returns valid, then *init* and *end* cannot cause the SHP, indeed if this is true and they are defined over the same window, then the SHP cannot occur for *init* and *end* events.

Iteration Event Patterns: when *init* and *end* events are iterations, the function returns *Valid* if $init.w = end.w$ (line 15), *ShpSimple(init, end)* = Valid (line 16) and *ShpPredicate(init.P, end.P)* = *Valid* (line 17). The Lemma 3.2.1 ensures that if *ShpSimple(init, end)* returns valid, then *init* and *end* cannot cause the SHP and this result can be easily extended for *ShpPredicate(init.P, end.P)* function, indeed, if these conditions are satisfied and *init* and *end* events are defined over the same window, then the SHP cannot occur for *init* and *end* events.

Proof of Theorem 3.2.1

We have to prove that if Algorithm 1 returns *Valid*, then *init* and *end* events cannot cause the SHP, if it returns *Invalid*, then they might cause the SHP, otherwise the SHP is avoided through *Post* procedure (line 8). Algorithm 1 returns *Valid* (lines 3, 5) in the following cases.

Simple Events: when *init* and *end* are simple events (line 2) and *ShpSimple* returns *Valid* (line 3). Lemma 3.2.1 ensures that if *ShpSimple(init, end)* returns valid, then *init* and *end* cannot cause the SHP.

Event Patterns: when *init* and *end* are event patterns (line 4) and *ShpPattern* returns *Valid* (line 5). Lemma 3.2.2 ensures that if *ShpPattern(init, end)* returns valid, then *init* and *end* cannot cause the SHP.

Algorithm 1 returns *Invalid* (lines 3, 5) in the following cases.

Simple Events: when *init* and *end* are simple events (line 2) and *ShpSimple* returns *Invalid* (line 3). Lemma 3.2.1 ensures that if *ShpSimple(init, end)* returns invalid, then *init* and *end* might cause the SHP.

Event Patterns: when *init* and *end* are event patterns (line 4) and *ShpPattern* returns *Invalid* (line 5). Lemma 3.2.2 ensures that if *ShpPattern(init, end)* returns invalid, then *init* and *end* might cause the SHP.

When *init* and *end* events do not fall in one of the previously mentioned cases, Algorithm 1 executes the *Post* procedure (line 8). In this case, the SHP is avoided by the *Post Processing* module which performs one of the previously described actions: (i) discard tuples causing the SHP or (ii) discard only the tuple causing the end of the emergency. In case (i) the emergency is not created, whereas in case (ii) the emergency is activated and never deactivated, thus in both cases the simultaneous creation and deletion of an emergency instance cannot occur.

3.2.2 Emergency Policy Administration

Emergency management is a complex task that we believe requires distributing the rights of create/modify emergency policies among different subjects, called *emergency managers*. Indeed, people in charge of the planning of response activities for emergency situations have a strong expertise in issues dealing with the particular field originating the emergency. For example, in the hospital scenario the head of cardiology ward has the best profile to indicate which activities have to be performed for cardiology emergencies, but not to determine the response plan for a breathing emergency. Moreover, relevant damage might be caused by emergency managers which, due to low expertise in IT security issues, might define dangerous emergency policies, as they might accidentally or maliciously write policies allowing unnecessary access to sensitive information or performing unnecessary obligations. For these reasons, we introduce administration policies that specify who are the emergency managers authorized to create/modify policies for which emergencies, and which tacp templates they are authorized to specify in emergency policies. For example, by means of administration policies, we should be able to state that the pediatric ward administrator is authorized to define tacp templates only on EMRs of patients with age lower than 12, as well as that a generic doctor is authorized to define emergencies only over the VitalSigns data stream and not on the BankTransaction stream, because is not within his/her competences. In order to enforce the above-described restrictions, we make use of the concepts of *emergency scope* and *tacp scope*, which are formally defined as follows.

Definition 3.2.4 ***(Emergency Scope):*** *An* emergency scope *is a tuple (*event*,* streams*,* operators*), where* event ∈ *{*`init`*,* `end`*,* `both`*},* streams *is a set of stream names, and* operators *is a set of CESL operators.*

Given an emergency description e and an emergency scope *emg_scope*, we say that e is valid w.r.t. *emg_scope*, if the *init* (*end* or *both*, respectively) event is defined on a subset of the streams specified in *emg_scope.streams*, by using a subset of CESL operators specified in *emg_scope.operators.*

Definition 3.2.5 ***(Tacp Scope):*** *A* tacp scope *is a tuple (*sbj*,* obj*,* priv*,* ctx*,* obl*) where:* sbj*,* obj *and* ctx *are subject, object and context specification, respectively;* priv *and* obl *are a set of allowed privileges and actions, respectively.*

Given a tacp template *tacp* and a tacp scope *tacp_scope*, we say that *tacp* is valid w.r.t. a *tacp_scope* if: the subject (object, respectively) specification of *tacp* identifies a subset of subjects (objects, respectively) identified by *tacp_scope.sbj* (*tacp_scope.obj*, respectively), the set of values for a context attribute identified by a context specification of a *tacp* is a subset of the values identified by *tacp_scope.ctx* and the privileges (obligations) in *tacp.priv* (*tacp.obl*) is a subset of those privileges (obligations, respectively) identified in *tacp_scope.priv* (*tacp_scope.obl*). Based on emergency and tacp scopes, we can now formalize the emergency administration policies, as follows.

Definition 3.2.6 ***(Emergency Administration Policy):*** *An emergency administration policy is a tuple (*admin_sbj*,* emg_scope*,* tacp_scope*,* obl*), where* admin_sbj *denotes the users authorized to specify emergency policies* *whose emergency description is valid w.r.t.* emg_scope *and whose tacp template is valid w.r.t* tacp_scope, *and containing only a subset of obligations listed in* obl.

Example 3.2.9 *Let us assume that in our reference scenario the following administration policy is defined: (*CardiologyAdministrator, VitalSignsScope, CardiologyScope, *{*call_ambulance*}), where the only allowed emergency obligation is* call_ambulance, *whereas* VitalSignsScope *and* CardiologyScope *are specified as follows.*

```
VitalSignsScope {
   (Both,(VitalSigns,any));
}

CardiologyScope {
  sbj: (doctor, ward=cardiology);
  obj: (EMR, ward=cardiology);
  priv: {read};
  ctx: -;
  obl: {mailto};
}
```

VitalSignsScope *denotes emergencies where both* init *and* end *events are defined only over the* VitalSigns *stream (by using any of the CESL operators). In contrast, the tacp scope includes subjects with the doctor role who belong to the cardiology ward, and objects corresponding to the EMR of patients of the cardiology ward. Moreover, the only allowed privilege and obligation are, respectively,* read *and* mailto. *Consider the emergency policy defined in Example 3.2.2. This is not valid w.r.t. the above defined administration policy because the tacp is not valid w.r.t.* CardiologyScope, *since subjects identified by* HypertensionPolicy *include users with the paramedic role and not with the doctor role. The* HypertensionPolicy *can be easily modified to be valid w.r.t. the* CardiologyScope *changing the subject specification as (doctor, doctor_id = call.doctor_id). In this case* HypertensionPolicy *is valid after rewriting and the rewritten tacp is the following. More details about tacp rewriting are provided in the following Section.*

```
HypertensionPolicy {
  sbj: (doctor, doctor_id = call.doctor_id ∧
          ward = cardiology);
  obj: (EMR, patient_id = emg.patient_id ∧
          ward = cardiology);
  priv: read;
  ctx: -;
  obl: mailto(patient_mail);
}
```

Administration Policy Enforcement

The enforcement of emergency administration policies is carried out each time a user defines or modifies an emergency policy, with the aim of verifying whether the new policy satisfies at least an administration policy. In case an emergency policy is not valid w.r.t. the specified administration policies, a set of rewriting strategies are applied, aiming to re-define the invalid emergency policy so as to make it valid w.r.t. at least one of the specified administration policies. Every time an emergency policy is rewritten, a warning is sent to the emergency manager who has defined the policy in order to inform him about the rewriting operation and, in case of bad rewriting, to manually correct the policy. In case an emergency policy is not valid w.r.t. any administration policy and rewriting is not possible, the emergency policy is discarded and the policy issuer is warned. When a user defines/modifies an emergency policy, the validity of the new emergency policy is verified by Algorithm 2.

Algorithm 2: ValidateEmergencyPolicies

```
   input  : $ep$, the new emergency policy to be validated
   input  : $u$, the user which is trying to define $ep$
   output : $ep$, $\emptyset$ or a list of valid rewritten emergency policies
 1 Let $EAPR$ be the Emergency Administration Policy Base;
 2 rwEPs = $\emptyset$;
 3 foreach $eap \in EAPR$ do
 4     <r, np> = CheckEmergencyPolicy (u, ep, eap);
 5     if $r$ = Valid then  return $ep$;
 6     if $r$ = ValidAfterRw then
 7         rwEPs = rwEPs $\cup$ {np};
 8         Warn (u, ep, eap);
 9 end
10 if $rwEPs = \emptyset$ then  Warn (u, ep);
11 return rwEPs;
```

Algorithm 2 takes as input an emergency policy *ep* and the user u who is trying to define it. Algorithm 2 checks *ep* against each administration policy *eap* in the Emergency Administration Policy Repository $EAPR$ (lines 3-9) by using the *CheckEmergencyPolicy* function (line 4). This function takes as input u, *ep*, *eap* and returns a pair *<r, np>* with one of the following values: *<Valid, ep>*, if *ep* is valid w.r.t. *eap*, *<Invalid, ∅>*, if *ep* is not valid w.r.t. *eap*, *<ValidAfterRw, np>*, where *np* is a rewritten emergency policy, if *ep* is not valid but the rewriting strategy can be applied. If r = *Valid*, then Algorithm 2 returns *ep* (line 5). If r = *ValidAfterRw*, then the rewritten emergency policy *np* is stored into the *rwEPs* set (line 7) and user *u* is informed that the emergency policy he/she has defined has been rewritten (line 8). In case *ep* is not valid, when Algorithm 2 has analyzed all the emergency administration policies, it returns *rwEPs*, which could be empty or contain the set of rewritten emergency policies (line 11). In case *rwEPs* is empty, the *ep* emergency policy is not inserted into the policy base and the user *u* is warned about the wrong definition of *ep* (line 10).

Function CheckEmergencyPolicy(u, ep, eap)

```
1  Let np = (tacp, emg, obl) be initialized empty;
2  EmgChk = ChkEmgScope ($ep.emg$, $eap.emg\_scope$);
3  <r, np.tacp> = RwTacp ($ep.tacp$, $eap.tacp\_scope$);
4  if $u \in eap.admin\_sbj \wedge ep.obl \subseteq eap.obl \wedge EmgChk = true \wedge r =$ Valid then
5      return <Valid, $ep$>;
6  if $u \notin eap.admin\_sbj \vee ep.obl \cap eap.obl = \emptyset \vee EmgChk = false \vee r =$ Invalid
   then
7      return <Invalid, $\emptyset$>;
8  np.emg = $ep.emg$;
9  np.obl = $ep.obl \cap eap.obl$;
10 return <ValidAfterRw,np>;
```

This function first checks if the emergency description *ep.emg* is valid w.r.t. the emergency scope *eap.emg_scope* (line 2) through function *ChkEmg Scope* (see its description later on). Then, it calls function *RwTacp* (line 3), which takes as arguments the tacp template contained into the input emergency policy and the tacp scope of the input administrative policy and returns a pair *<r, np.tacp>* that can have one of the following values: *<Valid, ep.tacp>*, if *ep.tacp* is valid w.r.t. *eap.tacp_scope*, *<Invalid, ∅>*, if *ep.tacp* is not valid w.r.t. *eap.tacp_scope*, *<ValidAfterRw, np.tacp>*, where *np.tacp* is a rewritten tacp, if *ep.tacp* is not valid w.r.t. *eap.tacp_scope*, but it can be rewritten into the valid policy *np.tacp*. Then, *CheckEmergencyPolicy* verifies whether the user is among the authorized users in *eap*, obligations specified in *ep* are a subset of those authorized in *eap* and both the tacp template and the emergency description contained into the input emergency policy are valid w.r.t. the corresponding scope (line 4). If all these conditions are satisfied, then *CheckEmergencyPolicy* returns *< Valid*, *ep>* (line 5), otherwise it checks if there is at least a condition to consider *ep* not rewritable into a valid policy, that is, if *u* is not among the authorized users in *eap*, or obligations required in *ep* are disjoint from obligations allowed in *eap*, or *CheckEmergencyScope* returns false or *RwTacp* returns *Invalid* (line 6). If at least a condition holds, then *CheckEmergencyPolicy* returns *<Invalid*, ∅*>* (line 7). Otherwise, the rewriting is possible, thus *CheckEmergencyPolicy* returns *< ValidAfterRw*, *np>* (line 10), where *np* is the emergency policy resulting from the *tacp* rewriting performed by *RwTacp*, whereas obligations are given by the intersection between those required in *ep* and those authorized in *eap* (line 9).

Function ChkEmgScope(emg,scope)

```
1  switch $scope.event$ do
2      case $init$
3          if GetStreams $(emg.init) \not\subseteq scope.streams$ then
4              return false;
5          if GetOperators $(emg.init) \not\subseteq scope.operators$ then
6              return false;
7      case $end$
8          if GetStreams $(emg.end) \not\subseteq scope.streams$ then
9              return false;
10         if GetOperators $(emg.end) \not\subseteq scope.operators$ then
11             return false;
12     otherwise if GetStreams $(emg.init) \not\subseteq scope.streams \wedge$ GetStreams $(emg.end) \not\subseteq scope.streams$ then
13         return false;
14     if GetOperators $(emg.init) \not\subseteq scope.operators \wedge$ GetOperators $(emg.end) \not\subseteq scope.streams$ then
15         return false;
16 endsw
17 return true;
```

The *ChkEmgScope* function takes as input an emergency *emg* and an emergency scope *scope* and returns true or false whether *emg* is valid or not w.r.t. *scope*. The function considers three cases depending on the content of the field *scope.event* (line 1). If $scope.event = init$ (line 2), then the function checks if the streams over which init is defined are not a subset of the streams contained in the scope (line 3) and if the operators used in init specification are not a subset of the scope operators (line 5), in case one of these checks succeeds the function return false, i.e., *emg* is not valid w.r.t. *scope*. The two functions *GetStreams* and *GetOperators* return respectively the list of streams and operators over which the event passed as argument is defined. In a similar way, the function checks streams and operators for the other cases ($scope.event = end$, line 7 and $scope.event = both$, line 12). In case all these checks fail, then *ChkEmgScope* function returns true and the emergency *emg* is valid w.r.t. *scope* (line 17).

Function RwTacp(t,s)

```
1  Let n = (sbj, obj, ctx, obl) be initialized empty;
2  rw = false;
3  <res, np.sbj> = RwTacpSbj (t.sbj,s.sbj);
4  if res = Invalid then return <Invalid, $\emptyset$>;
5  if res = ValidAfterRw then  rw = true;
6  <res, np.obj> = RwTacpObj (t.obj,s.obj);
7  if res = Invalid then return <Invalid, $\emptyset$>;
8  if res = ValidAfterRw then rw = true;
9  if $t.priv \cap s.priv = \emptyset$ then return <Invalid, $\emptyset$>;
10 if $t.priv \not\subseteq s.priv$ then
11     rw = true;
12     np.priv = $t.priv \cap s.priv$;
13 end
14 <res, np.ctx> = RwTacpExp (t.ctx,s.ctx);
15 if res = Invalid then return <Invalid, $\emptyset$>;
16 if res = ValidAfterRw then rw = true;
17 if $t.obl \cap s.obl = \emptyset$ then return <Invalid, $\emptyset$>;
18 if $t.obl \not\subseteq s.obl$ then
19     rw = true;
20     np.obl = $t.obl \cap s.obl$;
21 end
22 if rw = false then  return <Valid, t>;
23 else  return <ValidAfterRw, np>;
```

This function takes as input a tacp template t and a tacp scope s and returns a pair *<r, np>* whose possible values have been explained before. Function *RwTacp* checks the subject specification *t.sbj* against the subject specification *s.sbj* of the input tacp scope. This is done through function *RwTacpSbj*, which will be described later on (line 3). This function returns a pair *<res, np.sbj>*. If *res = Invalid*, then *RwTacp* returns *Invalid* (line 4). If *res = ValidAfterRw*, then the *rw* flag is set to true (line 5). Then, *RwTacp* checks the validity of the other tacp fields (i.e., the object specification and context condition) using functions *RwTacpObj* (line 6) and *RwTacpExp* (line 14), respectively. In contrast privileges and obligations validity are checked by computing their intersections with privileges/obligations authorized in the input scope. If the intersection is empty (lines 9, 17), then *RwTacp* returns *Invalid*. If tacp privileges (obligations, respectively) are not a subset of tacp scope ones (lines 10, 18), then a rewriting is possible (lines 11, 19) and the new privileges (obligations, respectively) are calculated (lines 12, 20) as the intersection of tacp and tacp scope privileges (obligations, respectively). When *RwTacp* has analyzed all the tacp fields, then, if none of them has been rewritten, the function returns the couple *<Valid, t>* (line 22), otherwise, it returns *<ValidAfterRw, np>*, where *np* is the rewritten tacp (line 23). In the following, we explain *RwTacpSbj*. *RwTacpObj* and *RwTacpExp* are not explained because they are similar to *RwTacpSbj*.

Function RwTacpSbj(s, t)

```
1 <RolesRes,np.roles> = RwRoles (s.roles, t.roles);
2 <CondRes,np.cond> = RwConditions (s.cond, t.cond);
3 if $RolesRes = Valid \wedge CondRes = Valid$ then
4     return <Valid, $t$>;
5 if $RolesRes = Invalid \vee CondRes = Invalid$ then
6     return <Invalid, $\emptyset$>;
7 else
8     return <ValidAfterRw, $np$>;
9 end
```

Function *RwTacpSbj* takes as input two subject specifications: s, the subject specification of a tacp scope and t the subject specification of a tacp. According to the adopted access control model, these subject specifications are pairs (*roles*, *cond*), where *roles* is a set of roles and *cond* is a Boolean expression on user profile attributes. First, *RwTacpSbj* verifies if roles and conditions specified in t satisfy the restrictions contained in s. This is done by two functions, namely, *RwRoles* (line 1) and *RwConditions* (line 2) that both return a pair *<r, np>*. If both roles and condition in t are valid w.r.t. s (line 3), then *RwTacpSbj* returns *<Valid, t>* (line 4). If at least one between roles and condition in tacp t is invalid w.r.t. s (line 5), then *RwTacpSbj* returns *<Invalid, ∅>* (line 6). Otherwise (line 7), *RwTacpSbj* returns *<ValidAfterRw, np>* (line 8), where *np* is the rewritten subject specification.

Function RwRoles

This function checks if roles in $t.roles$ are a subset of roles in $s.roles$, then it returns *<Valid, t.roles>*, if the two sets are disjoint, then it returns *<Invalid, ∅>*, otherwise, a rewriting is possible and the rewritten roles are calculated as the intersection between scope and tacp roles.

Function RwConditions(s.cond, t.cond)

```
1  rw = false;
2  validCond = false;
3  DNF_s = DNF (s.cond);
4  DNF_t = DNF (t.cond);
5  disj = $\emptyset$;
6  foreach conjunctive clause $c_h \in DNF\_s$ do
7      foreach conjunctive clause $c_k \in DNF\_t$ do
8          conj = $\emptyset$;
9          validClause = true;
10         foreach predicate $p_i \in c_h$ do
11             foreach predicate $p_j \in c_k$ do
12                 Let Att(p) the attribute over which predicate p is defined;
13                 if $Att(p_i) \neq Att(p_j)$ then
14                     rw = true;
15                     Insert $(p_i \wedge p_j)$ Into conj;
16                 else
17                     Let $VS\_p$ be the validity set of predicate $p$;
18                     if $VS\_p_i \cap VS\_p_j = \emptyset$ then
19                         validClause = false;
20                     else
21                         if $VS\_p_j \subseteq VS\_p_i$ then
22                             Insert $p_j$ Into conj;
23                         else
24                             rw = true;
25                             Insert $(p_i \wedge p_j)$ Into conj;
26                         end
27                     end
28                 end
29             end
30         end
31         if validClause = true then validCond = true;
32         Insert conj Into np.cond;
33     end
34 end
35 if validCond = false then
36     return <Invalid, $\emptyset$>;
37 else
38     if rw = false then return <Valid, t>;
39     else return <ValidAfterRw, np.cond>;
40 end
```

Function *RwConditions* takes as input two Boolean expressions: *s.cond* of a tacp scope s and *t.cond* of a tacp t. *RwConditions* first transforms both the expressions in Disjunctive Normal Form (DNF), i.e., a disjunction of conjunctive clauses, respectively *DNF_s* (line 3) and *DNF_t* (line 4). It then compares each conjunctive clause $c_h \in DNF_s$ against each conjunctive clause $c_k \in DNF_t$ (lines 6-34). This is performed analyzing each pair of predicates $p_i \in c_h$, $p_j \in c_k$ in the conjunctive clauses (lines 10-30). The predicates comparison depends on attributes over which they are defined. If p_i and p_j are not defined over the same attributes (line 13), then they must be rewritten (line 14) and inserted into the resulting conjunctive clause *conj* (line 15). If p_i and p_j are defined over the same attributes (line 16) then, the intersection of their validity sets[10] VS_p_i and VS_p_j is computed. If the intersection is empty (line 18), then variable *validClause* is set to false (line 19), i.e., the clause is invalid. Otherwise, if $VS\_p_j \subseteq VS\_p_i$ (line 21), then p_j is inserted into *conj* without rewriting it (line 22). Otherwise (line 23), a rewriting is needed (line 24). The rewriting is performed inserting $p_i \wedge p_j$ into *conj* (line 25). Once all pairs of predicates have been compared, then the conjunctive clause *conj* is inserted into the resulting disjunctive clause *np.cond* (line 32). Moreover, the *validClause* variable is checked, if its value is true, it means that all predicates in c_h are valid w.r.t. all predicates in c_k, thus *validCond* is set to true (line 31), i.e., the whole condition is valid. This is because in a disjunctive clause, if one clause is valid, then the entire condition is valid too. When *RwConditions* has analyzed all conditions: if all clauses are invalid (line 35), i.e., *valCond = false*, then it returns *<Invalid, ∅>* (line 36); if none of them has been rewritten, then it returns *<Valid, t>* (line 38); otherwise it returns *<ValidAfterRw, np.cond>* (line 39), where *np.cond* is the rewritten condition. Theorem 3.2.2 proves the correctness of Algorithm 2.

In the following, a temporary access control policy rewriting example is presented.

Example 3.2.10 ***(Emergency Policy Rewriting).*** *Consider the* HypertensionPolicy *presented in Example 3.2.1. This policy can be easily modified to be valid w.r.t. the* CardiologyScope *presented in Example 3.2.9. In this case* HypertensionPolicy *is valid after rewriting and the rewritten tacp is the following.*

```
HypertensionPolicy {
    sbj: (doctor, doctor_id = call.doctor_id ∧
            ward = cardiology);
    obj: (EMR, patient_id = emg.patient_id ∧
            ward = cardiology);
    priv: read;
    ctx: -;
    obl: mailto(patient_mail);
}
```

[10] The *validity set* of a predicate p is the set of values satisfying p.

Administration Correctness

Theorem 3.2.2 ***(ValidateEmergencyPolicies Correctness).*** *Let ep be an emergency policy submitted by a user u, and rwEPs be the set of authorized emergency policies returned by Algorithm 2. For each authorized emergency policy rwep $\in$ rwEPs, there exists at least an administration policy eap $\in$ EAPR (the Emergency Administration Policy Repository) such that eap authorizes rwep. If rwEPs $= \emptyset$, then none of the administration policies eap $\in$ EAPR authorizes ep, and ep cannot be rewritten in order to be valid w.r.t. at least one administration policy. Before proving it, we need to introduce a lemma stating the correctness of function* CheckEmergencyPolicy*.*

Lemma 3.2.3 ***(CheckEmergencyPolicy Correctness).*** *Let u be a user which is trying to define an emergency policy ep and eap an emergency administrative policy. Let <r, n> be the result returned by* CheckEmergencyPolicy(u, ep, eap) *function. If r is (i)* Valid*, then ep is valid w.r.t. eap; (ii)* Invalid*, then ep is not valid w.r.t. eap; otherwise (iii), it means that the emergency policy ep has been rewritten as np which is valid w.r.t. eap.*

Before proving Theorem 3.2.2, we need to prove Lemma 3.2.3. Before proving it, we need to introduce lemmas stating the correctness of functions *ChkEmgScope*, *RwRoles*, *RwConditions*, *RwTacpSbj*, *RwTacpObj* *RwTacpExp*.

Lemma 3.2.4 ***(ChkEmgScope Correctness).*** *Let e be an emergency description and s an emergency scope. Let S be the set of emergency descriptions valid w.r.t. s. If $ChkEmgScope(e, s)$ returns true, then $e \in S$; $e \notin S$ otherwise.*

We prove the lemma by proving that if there exists an emergency $e \notin S$ such that $ChkEmgScope(e, s)$ returns true, then a contradiction arises. *ChkEmgScope* returns *true* in the following cases.

- *s.event = init*: in this case *ChkEmgScope* returns *true* if *init* is defined over a subset of streams defined in *s.streams* and operators in *init* are a subset of operators in *s.operators*. If both these conditions hold, then, according to Definition 3.2.4, the emergency description e is valid w.r.t. s, thus $e \in S$.

- *s.event = end*: in this case *ChkEmgScope* returns *true* if *end* is defined over a subset of streams defined in *s.streams* and operators in *end* are a subset of operators in *s.operators*. If both these conditions hold, then, according to Definition 3.2.4, the emergency description e is valid w.r.t. s, thus $e \in S$.

- *s.event* = *both*: in this case *ChkEmgScope* returns *true* if *init* and *end* are defined over a subset of streams defined in *s.streams* and operators in *init* and *end* are a subset of operators in *s.operators*. If all these conditions are met, then e is valid w.r.t. s, thus $e \in S$.

Lemma 3.2.5 ***(RwRoles Correctness).*** *Let t and s be a tacp and a tacp scope, respectively. Let $t.roles$ and $s.roles$ be the set of roles in the subject specification of t and s, respectively. We denote with $S_{t.roles}$ and $S_{s.roles}$ the set of subjects identified by $t.roles$ and $s.roles$, respectively. If* RwRoles(s.roles, t.roles) *returns:* Valid*, then $S_{t.roles} \subseteq S_{s.roles}$;* Invalid*, then $S_{t.roles} \cap S_{s.roles} = \emptyset$; otherwise (i.e.,* ValidAfterRw*), it means that $t.roles$ has been rewritten as $np.roles$ which is valid w.r.t. $s.roles$, i.e., assuming $S_{np.roles}$ be the set of subjects identified by $np.roles$, then $S_{np.roles} \subseteq S_{s.roles}$.*

We have to prove that if *RwRoles* returns **Valid**, then $S_{t.roles} \subseteq S_{s.roles}$, if it returns **Invalid**, then $S_{t.roles} \cap S_{s.roles} = \emptyset$, otherwise the rewritten role specification *np.roles* identifies a set of subjects $S_{np.roles} \subseteq S_{s.roles}$. *RwRoles* returns *Valid* if $t.roles$ is a subset of $s.roles$, indeed in this case $S_{t.roles} \subseteq S_{s.roles}$, it returns *Invalid* if the intersection between $t.roles$ and $s.roles$ is empty, thus $S_{t.roles} \cap S_{s.roles} = \emptyset$. In case of rewriting, the new set of roles *np.roles* is calculated as the intersection of $t.roles$ and $s.roles$, which identifies the subjects $S_{np.roles} = S_{t.roles} \cap S_{s.roles}$, therefore $S_{np.roles} \subseteq S_{s.roles}$.

Lemma 3.2.6 ***(RwConditions Correctness).*** *Let t and s be a tacp and a tacp scope, respectively. Let $t.cond$ and $s.cond$ be the conditions in the subject specification of t and s, respectively. We denote with $S_{t.cond}$ and $S_{s.cond}$ the set of subjects identified by $t.cond$ and $s.cond$, respectively. If* RwConditions(s.cond, t.cond) *returns:* Valid*, then $S_{t.cond} \subseteq S_{s.cond}$;* Invalid*, then $S_{t.cond} \cap S_{s.cond} = \emptyset$; otherwise (i.e.,* ValidAfterRw*), it means that $t.cond$ has been rewritten as $np.cond$ which is valid w.r.t. $s.cond$, i.e., assuming $S_{np.cond}$ be the set of subjects identified by $np.cond$, then $S_{np.cond} \subseteq S_{s.cond}$.*

We prove the Lemma by induction on the dimension of the DNF of *t.cond*, where by dimension we mean the number of conjunctive clauses and the number of predicates in each clause. Thus, proving the Lemma by induction implies to prove that the lemma holds for a DNF composed of one clause with one predicate (*Basis n=1*). Then, we assume that the Lemma holds for a DNF with one clause composed of n predicates and we prove that it holds also for DNF with one clause composed of $n + 1$ predicates. If the Lemma is demonstrated for a DNF with one conjunctive clause, then it is valid also for a DNF with more than one clause since in a disjunction, if one clause is valid, then the entire disjunction is valid too. In the demonstration, we assume that the DNF of *s.cond* has the following from $c_1 \vee \ldots \vee c_n$, where each conjunctive clause c_i has the from $p_1 \wedge \ldots \wedge p_m$. The set of subjects identified by *s.cond* is $S_{s.cond} = S_{c_1} \cup \ldots \cup S_{c_n}$ and the set of subjects identified by a conjunctive clause c_i is $S_{c_i} = S_{p_1} \cap \ldots \cap S_{p_m}$, where S_{c_j} and S_{p_j} are the set of subjects whose profiles satisfy conditions in clause c_j and predicate p_j.

Basis: The DNF of *t.cond* is composed of one predicate p, thus $S_{t.cond} = S_p$.

By construction, *RwConditions* returns **Valid** (line 38) when *validCond* = *true* (i.e., check at line 35 fails) and *rw* = *false.* These conditions are satisfied when there is at least one clause in *DNF(t.cond)* which is valid w.r.t. a clause in *DNF(s.cond)* (line 31) and none of the predicates has been rewritten. A clause c_1 is valid w.r.t. another clause c_2 if all predicates in c_1 are valid w.r.t. all predicates in c_2. More formally, these conditions are satisfied when $\exists\ c_h \in DNF(s.cond)$ (lines 6-34) such that $\forall\ p_i \in c_h$ (lines 10-30), $Att(p_i) = Att(p)$ (line 16) and $VS\_p \subseteq VS\_p_i$[11] (line 21). If these conditions are met, it means that the validity set of p is a subset of the validity sets of each p_i, i.e., the subjects identified by p are a subsets of subjects identified by $p_1, \ldots, p_n$, that is, $S_p \subseteq S_{p_1}, \ldots, S_p \subseteq S_{p_m}$. Since $S_{c_i} = S_{p_1} \cap \ldots \cap S_{p_m}$, S_p is also a subset of S_{c_i}. Moreover, since $S_{s.cond} = S_{c_1} \cup \ldots \cup S_{c_n}$, if $S_p \subseteq S_{c_1}, \ldots, S_p \subseteq S_{c_n}$, then $S_p \subseteq S_{s.cond}$.

RwConditions returns **Invalid** (line 36) when *validCond* = *false* (line 35). This condition is satisfied when all clauses in *DNF(t.cond)* are invalid w.r.t. all clauses in *DNF(s.cond)*. A clause c_1 is invalid w.r.t. another clause c_2 if at least one predicate in c_1 is invalid w.r.t. a predicate in c_2. More formally, these conditions are satisfied when $\forall\ c_h \in DNF(s.cond)$ (lines 6-34), $\exists\ p_i \in c_h$ (lines 10-30) such that $Att(p_i) = Att(p)$ (line 16) and $VS\_p_i \cap VS\_p = \emptyset$ (line 18). If this condition is met, it means that $S_p \cap S_{p_i} = \emptyset$. Since $S_{c_i} = S_{p_1} \cap \ldots \cap S_{p_m}$, if $S_p \cap S_{p_i} = \emptyset$, then $S_p \cap S_{c_i} = \emptyset$. If this is true for all clauses, then $S_{c_1} \cap S_p = \emptyset, \ldots, S_{c_n} \cap S_p = \emptyset$. Since $S_{s.cond} = S_{c_1} \cup \ldots \cup S_{c_n}$, then $S_p \cap S_{s.cond} = \emptyset$.

We recall that, in case *RwConditions* returns *ValidAfterRw*, the rewritten tacp subject specification has the following form: *np.cond* = $nc_1 \vee \ldots \vee nc_n$, where each conjunctive clause nc_i has the form: $np_1 \wedge \ldots \wedge np_m$. The set of subjects identified by *np.cond* is $S_{np.cond} = S_{nc_1} \cup \ldots \cup S_{nc_n}$ and the set of

[11] We recall that the validity set VS_p is defined as the set of tuples that satisfies the predicate p.

subjects identified by a conjunctive clause nc_i is $S_{nc_i} = S_{np_1} \cap \ldots \cap S_{np_m}$.

Function *RwConditions* returns **ValidAfterRw** (line 39) when $validCond = true$ (i.e., check at line 35 fails) and $rw = true$. These conditions are satisfied if there exists at least a predicate that has been rewritten and none of the other predicates is invalid w.r.t. p. Indeed, if this is the case, the function would stop, returning *Invalid*. If these conditions are met it means that a predicate p_i has been rewritten and $\forall$ p_j such that $p_j \neq p_i$, there are three possibilities.

1. $Att(p_j) \neq Att(p)$ (line 13), in this case a rewriting is performed as $np_j = p_j \wedge p$.
2. $Att(p_j) = Att(p)$ (line 16) and $VS\_p \subseteq VS\_p_j$ (line 21), in this case no rewriting is needed, therefore $np_j = p_j$.
3. $Att(p_j) = Att(p)$ (line 16) and $VS\_p \nsubseteq VS\_p_j$ (line 23), in this case a rewriting is performed as $np_j = p_j \wedge p$.

In conclusion, predicate p_i has been rewritten, thus $np_i = p_i \wedge p$, i.e., $S_{np_i} = S_{p_i} \cap S_p$, indeed $S_{np_i} \subseteq S_{p_i}$. Regarding any other predicate p_j, there are two possibilities (1) p_j is rewritten (case 1, 3) as $np_j = p_j \wedge p$, thus $S_{np_j} = S_{p_j} \cap S_p$ or (2) p_j is not rewritten (case 2), thus $S_{np_j} = S_{p_j}$. In both cases, $S_{np_j} \subseteq S_{p_j}$. In light of these considerations, a rewritten clause nc_i identifies a set of subjects $S_{nc_i} = S_{np_1} \cap \ldots \cap S_{np_m}$, where each S_{np_i} is a subset of S_{p_i}, thus $S_{nc_i} \subseteq S_{c_i}$. Since $S_{s.cond} = S_{c_1} \cup \ldots \cup S_{c_n}$ and $S_{np.cond} = S_{nc_1} \cup \ldots \cup S_{nc_n}$, then $S_{np.cond} \subseteq S_{s.cond}$.

Induction: Let us now assume that report holds for a subject specification s_t composed of one clause with n predicates. We prove the report for a subject specification $s_{t'}$, composed of one clause with $n+1$ predicates. Since the report holds for s_t, the set of subjects S_t identified by s_t is a subset of S_s (i.e., $S_t \subseteq S_s$). Adding a predicate to s_t, we obtain the following DNF: $s_{t'} = (p_1 \wedge \ldots \wedge p_n \wedge p_{n+1})$. Indeed, $s_{t'}$ is more restrictive than s_t, thus $S_{t'} \subseteq S_t$ and consequently $S_{t'} \subseteq S_s$, which prove the report.

Lemma 3.2.7 ***(RwTacpSbj Correctness).*** *Let t and s be a tacp and a tacp scope, respectively. We denote with S_t and S_s the set of subjects identified by t and s, respectively. If* RwTacpSbj(s, t) *returns* Valid*, then $S_t \subseteq S_s$;* Invalid*, then $S_t \cap S_s = \emptyset$; otherwise (i.e.,* ValidAfterRw*), it means that t has been rewritten as np which is valid w.r.t. s, i.e., assuming S_{np} be the set of subjects identified by np, then $S_{np} \subseteq S_s$.*

By construction, *RwTacpSbj(s, t)* returns **Valid** (line 4) if both *RwRoles(s.roles, t.roles)* and *RwConditions(s.cond, t.cond)* return *Valid* (line 3). Lemma 3.2.5 ensures that if *RwRoles* returns *Valid*, then $S_{t.roles} \subseteq S_{s.roles}$ and Lemma 3.2.6 ensures that if *RwConditions* returns *Valid*, then $S_{t.cond} \subseteq S_{s.cond}$. Since the set of subjects identified by a subject specification x is $S_x = S_{x.roles} \cap S_{x.cond}$, then if $S_{t.roles} \subseteq S_{s.roles}$ and $S_{t.cond} \subseteq S_{s.cond}$, then $S_t \subseteq S_s$.

The *RwTacpSbj(s, t)* function returns **Invalid** (line 6) if *RwRoles(s.roles, t.roles)* or *RwConditions(s.cond, t.cond)* returns *Invalid* (line 5). Lemma 3.2.5 ensures that if *RwRoles* returns *Invalid*, then $S_{t.roles} \cap S_{s.roles} = \emptyset$, similarly, Lemma 3.2.6 ensures that if *RwConditions* returns *Invalid*, then $S_{t.cond} \cap S_{s.cond} = \emptyset$. Thus, if $S_{t.roles} \cap S_{s.roles} = \emptyset$ or $S_{t.cond} \cap S_{s.cond} = \emptyset$, then $S_t \cap S_s = \emptyset$.

The *RwTacpSbj(s, t)* function returns **ValidAfterRw** (line 8) if at least one of the functions *RwRoles(s.roles, t.roles)* and *RwConditions(s.cond, t.cond)* returns *ValidAfterRw* and the other function does not return *Invalid* (e.g., conditions in lines 3 and 5 are not satisfied), i.e., the other function might return *Valid* or *ValidAfterRw*. Lemma 3.2.5 ensures that if *RwRoles* returns *ValidAfterRw*, then the subjects identified by *np.roles* are $S_{np.roles} \subseteq S_{s.roles}$ and Lemma 3.2.6 ensures that if *RwConditions* returns *ValidAfterRw*, then the subjects identified by *np.cond* are $S_{np.cond} \subseteq S_{s.cond}$. Indeed, if $S_{np.roles} \subseteq S_{s.roles}$ and $S_{np.cond} \subseteq S_{s.cond}$, then $S_{np} \subseteq S_s$.

We now introduce Lemmas 3.2.8 and 3.2.9 stating the correctness of *RwTacpObj* and *RwTacpExp* functions. Although the code of these functions is not presented in Section 4, we give the following two Lemmas because these functions are similar to *RwTacpSbj*, but we not provide the formal demonstrations.

Lemma 3.2.8 ***(RwTacpObj Correctness).*** *Let t and s be a tacp and a tacp scope, respectively. We denote with O_t and O_s the set of objects identified by t and s, respectively. If* RwTacpObj(s, t) *returns* Valid*, then $O_t \subseteq O_s$;* Invalid*, then $O_t \cap O_s = \emptyset$; otherwise (i.e.,* ValidAfterRw*), it means that t has been rewritten as np which is valid w.r.t. s, i.e., assuming O_{np} be the set of objects identified by np, then $O_{np} \subseteq O_s$.*

Lemma 3.2.9 ***(RwTacpExp Correctness).*** *Let t and s be a tacp and a tacp scope, respectively. We denote with C_t and C_s the set of values for a context attribute a identified by the context expressions in t and s, respectively.*[12]

[12] We recall that in our model a context is a set of pairs (a, v), where a is a context

If RwTacpExp(s, t) *returns* Valid*, then* $C_t \subseteq C_s$*;* Invalid*, then* $C_t \cap C_s = \emptyset$*; otherwise (i.e.,* ValidAfterRw*), it means that t has been rewritten as np which is valid w.r.t. s, i.e., assuming* C_{np} *be the set of values for the context attribute a identified by the context expression np, then* $C_{np} \subseteq C_s$*.*

Lemma 3.2.10 ***(RwTacp Correctness).*** *Let t and s be a tacp and a tacp scope, respectively. Let <r, np> be the result returned by* RwTacp(t,s) *function. If r is (i)* Valid*, then t is valid w.r.t. s; (ii)* Invalid*, then t is not valid w.r.t. s; otherwise (iii), it means that the tacp t has been rewritten as np which is valid w.r.t. s.*

By Construction the *RwTacp(t,s)* function returns **Valid** if variable *rw* is false (line 22). This becomes true if *RwTacpSbj* (line 3), *RwTacpObj* (line 6), *RwTacpExp* (line 14) return *Valid* and *t.priv* $\subseteq$ *s.priv* and *t.obl* $\subseteq$ *s.obl* (i.e., checks at lines 10 and 18 fail). Lemma 3.2.7 (3.2.8 and 3.2.9, respectively) ensures that if *RwTacpSbj* (*RwTacpObj* and *RwTacpExp*, respectively) returns *Valid*, then $S_{t.sbj} \subseteq S_{s.sbj}$ ($O_{t.obj} \subseteq O_{s.obj}$ and $C_{t.ctx} \subseteq C_{s.ctx}$, respectively). According to Definition 3.2.5, a tacp t is valid w.r.t. a tacp scope s if (i) the set of subjects identified by the subject specification *t.sbj* is a subset of the set of subjects identified by *s.sbj*, i.e., $S_{t.sbj} \subseteq S_{s.sbj}$, (ii) the set of objects identified by the object specification *t.obj* is a subset of the set of objects identified by *s.obj*, i.e., $O_{t.obj} \subseteq O_{s.obj}$, (iii) the privileges in *t.priv* are a subset of the privileges in *s.priv*, i.e., *t.priv* $\subseteq$ *s.priv*, (iv) the set of values for a context attribute a identified by the context expressions *t.ctx* is a subset of the set of values identified by *s.ctx* for the same attribute, i.e., $C_{t.ctx} \subseteq C_{s.ctx}$, (v) the obligations in *t.obl* are a subset of the obligations in *s.obl*, i.e., *t.obl* $\subseteq$ *s.obl*. Lemmas 3.2.7, 3.2.8 and 3.2.9 ensure that properties (i-ii-iv) are all satisfied when *RwTacp(t,s)* function returns *Valid*, whereas properties (iii-v) are satisfied when *RwTacp(t,s)* returns *Valid* because it returns valid only if *t.priv* $\subseteq$ *s.priv* and *t.obl* $\subseteq$ *s.obl*.

The *RwTacp(s, t)* function returns **Invalid** (lines 4, 7, 9, 15, 17) if one function among *RwTacpSbj* (line 3), *RwTacpObj* (line 6) and *RwTacpExp* (line 14) returns *Invalid* (lines 4, 7, 15) or in case *t.priv* $\cap$ *s.priv* $= \emptyset$ (line 9) or in case *t.obl*$\cap$*s.obl* $= \emptyset$ (line 17). Lemma 3.2.7 (3.2.8 and 3.2.9, respectively) ensures that if *RwTacpSbj* (*RwTacpObj* and *RwTacpExp*, respectively) returns *Invalid*, then $S_{t.sbj} \cap S_{s.sbj} = \emptyset$ ($O_{t.obj} \cap O_{s.obj} = \emptyset$ and $C_{t.ctx} \cap C_{s.ctx} = \emptyset$, respectively). According to Definition 3.2.5, a tacp t is not valid w.r.t. a tacp scope

attribute and v is its current value and a context expression over a represents a set of values V for a; if $v \in V$, then the context expression is satisfied.

s if one of the following conditions is satisfied: (1) the set of subjects identified by the subject specification $t.sbj$ is disjoint from set of subjects identified by $s.sbj$, i.e., $S_{t.sbj} \cap S_{s.sbj} = \emptyset$, (2) the set of objects identified by the object specification $t.obj$ is disjoint from the set of objects identified by $s.obj$, i.e., $O_{t.obj} \cap O_{s.obj} = \emptyset$, (3) the privileges in $t.priv$ are disjoint from the privileges in $s.priv$, i.e., $t.priv \cap s.priv = \emptyset$, (4) the set of values for a context attribute a identified by the context expression $t.ctx$ is disjoint from the set of values identified by $s.ctx$ for the same attribute, i.e., $C_{t.ctx} \cap C_{s.ctx} = \emptyset$, (5) the obligations in $t.obl$ are disjoint from the obligations in $s.obl$, i.e., $t.obl \cap s.obl = \emptyset$. Lemmas 3.2.7, 3.2.8 and 3.2.9 ensure that one of the properties (1-2-4) is satisfied in case *RwTacp(s, t)* function returns *Invalid*. Moreover, if one of the conditions $t.priv \cap s.priv = \emptyset$ and $t.obl \cap s.obl = \emptyset$ is satisfied, then the *RwTacp(s, t)* function returns *Invalid*.

The *RwTacp(t,s)* function returns **ValidAfterRw** (line 23) if variable *rw* is true (i.e., check at line 22 fails). This becomes true if one of the functions *RwTacpSbj* (line 3), *RwTacpObj* (line 6) and *RwTacpExp* (line 14) returns *ValidAfterRw* (lines 5, 8, 16) or, assuming *np* is the rewritten tacp, when $np.priv \not\subseteq s.priv$ (line 10) and $np.obl \not\subseteq s.obl$ (line 18). Assuming *np* is the rewritten tacp, Lemma 3.2.7 (3.2.8 and 3.2.9, respectively) ensures that if *RwTacpSbj* (*RwTacpObj* and *RwTacpExp*, respectively) returns *ValidAfterRw*, then $S_{np.sbj} \subseteq S_{s.sbj}$ ($O_{np.obj} \subseteq O_{np.obj}$ and $C_{np.ctx} \subseteq C_{s.ctx}$, respectively). According to Definition 3.2.5, the rewritten tacp *np* is valid w.r.t. a tacp scope *s* if conditions (i-v) are all satisfied for the tacp *np*. Lemmas 3.2.7, 3.2.8 and 3.2.9 ensure that properties (i-ii-iv) are all satisfied when *RwTacp(t,s)* function returns *ValidAfterRw*, whereas properties (iii-v) are satisfied when *RwTacp(t,s)* returns *ValidAfterRw* because the rewritten privileges are calculated as $np.priv = t.priv \cap s.priv$ (line 11), thus $np.priv \subseteq s.priv$ and the rewritten obligations are calculated as $np.obl = t.obl \cap s.obl$ (line 19), thus $np.obl \subseteq s.obl$.

Proof of Lemma 3.2.3

By Construction, function *CheckEmergencyPolicy* returns **Valid** (line 5) if $u \in eap.admin\_sbj$, $ep.obl \subseteq eap.obl$, *ChkEmgScope(ep.emg, eap.emg_ scope)* returns *true* and *RwTacp(ep.tacp, eap.tacp_ scope)* returns *Valid* (line 4). According to Definition 3.2.6, an administrative policy *eap* authorizes an emergency policy *ep* (i.e., *ep* is valid w.r.t. *eap*), if the following conditions are all satisfied: (i) the *ep* issuer *u* belongs to subjects in *eap.admin_sbj*, (ii) the set of obligations in *ep.obl* is a subset of the obligations in *eap.obl*, (iii) the emergency *ep.emg* is valid w.r.t. the emergency scope *eap.emg_scope* and (iv) the tacp *ep.tacp* is valid w.r.t. the tacp scope *eap.tacp_scope*. Regarding condition (i), obviously if $u \in$ *eap.admin_sbj*, then it means that the *ep* issuer u belongs to subjects in *eap.admin_sbj*. Regarding condition (ii), obviously if *ep.obl* $\subseteq$ *eap.obl*, then the set of obligations in *ep.obl* is a subset of the obligations in *eap.obl*. Regarding condition (iii), Lemma 3.2.4 ensures that if *CheckEmergencyPolicy(ep.emg, eap.emg_scope)* returns *true*, then *ep.emg* is valid w.r.t. *eap.emg_scope*. Regarding condition (iv), Lemma 3.2.10 ensures that if *RwTacp(ep.tacp, eap.tacp_scope)* returns *Valid*, then *ep.tacp* is valid w.r.t. *eap.tacp_scope*. In light of these considerations, the *CheckEmergencyPolicy* function returns *Valid*, when conditions (i-iv) are all satisfied, i.e., when the administrative policy *eap* authorizes *ep*.

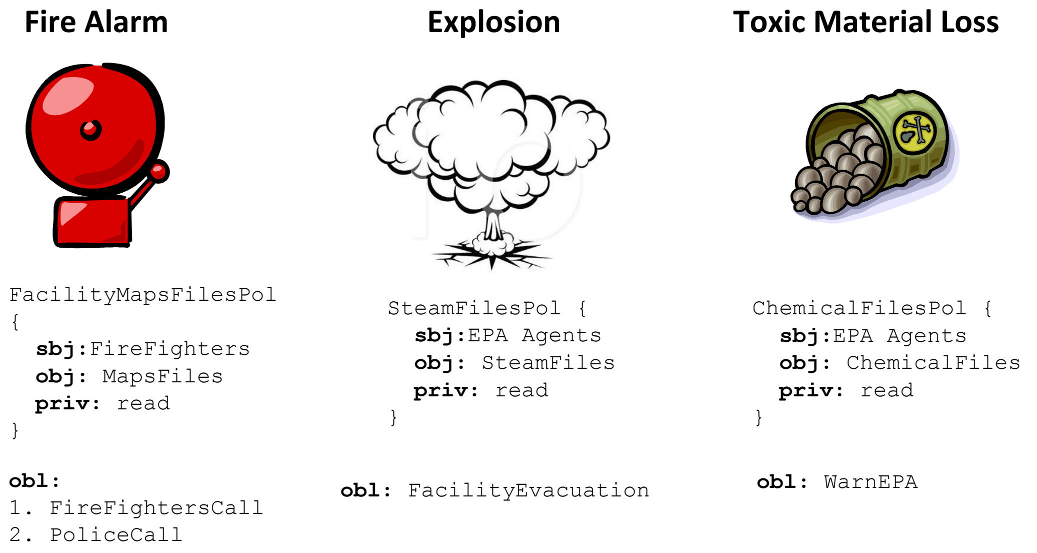
Function *CheckEmergencyPolicy* returns **Invalid** (line 7) if $u \notin$ *eap.admin_sbj* or *ep.obl* $\cap$ *eap.obl* $= \emptyset$ or *ChkEmgScope(ep.emg, eap.emg_scope)* returns *false* or *RwTacp(ep.tacp, eap.tacp_scope)* returns *Invalid* (line 6). According to Definition 3.2.6, an administrative policy *eap* does not authorizes an emergency policy *ep*, if one of the following conditions is satisfied: (1) the *ep* issuer u does not belongs to subjects in *eap.admin_sbj*, (2) the set of obligations in *ep.obl* is disjoint from the set of obligations in *eap.obl*, (3) the emergency *ep.emg* is not valid w.r.t. the emergency scope *eap.emg_scope* and (4) the tacp *ep.tacp* is not valid w.r.t. the tacp scope *eap.tacp_scope*. Regarding condition (1), obviously if $u \notin$ *eap.admin_sbj*, then it means that the *ep* issuer u belongs to subjects in *eap.admin_sbj*. Regarding condition (2), obviously if *ep.obl* $\cap$ *eap.obl* $= \emptyset$, then the set of obligations in *ep.obl* is disjoint from the set of obligations in *eap.obl*. Regarding condition (3), Lemma 3.2.4 ensures that if *ChkEmgScope(ep.emg, eap.emg_scope)* returns *false*, then *ep.emg* is not valid w.r.t. *eap.emg_scope*. Regarding condition (4), Lemma 3.2.10 ensures that if *RwTacp(ep.tacp, eap.tacp_scope)* returns *Invalid*, then *ep.tacp* is not valid w.r.t. *eap.tacp_scope*. In light of these considerations, the *CheckEmergencyPolicy* function returns *Invalid*, when at least one condition in (i-iv) is satisfied, i.e., when the administrative policy *eap* does not authorizes *ep*.

Function *CheckEmergencyPolicy* returns **ValidAfterRw** (line 10) if *np* is valid w.r.t. *eap*. The *np* policy is calculated as *np.emg* = *ep.emg* (line 8), *np.obl* = *ep.obl* $\cap$ *eap.obl* (line 9) and *np.tacp* is the rewritten tacp returned by the *RwTacp* function (line 3). According to Definition 3.2.6, an administrative policy *eap* authorizes an emergency policy *np* (i.e., *np* is valid w.r.t. *eap*), if conditions (i-v) are all satisfied. Regarding condition (i), obviously if $u \in$ *eap.admin_sbj*, then it means that the *np* issuer u belongs to subjects in *eap.admin_sbj*. Regarding condition (ii), obviously *np.obl* $\subseteq$ *eap.obl* because it is calculated as *np.obl* = *ep.obl* $\cap$ *eap.obl*. Regarding condition (iii), Lemma 3.2.4 ensures that if *ChkEmgScope(np.emg, eap.emg_ scope)* returns *true*, then *np.emg* is valid w.r.t. *eap.emg_scope*. Regarding condition (iv), Lemma 3.2.10 ensures that if *RwTacp(np.tacp, eap.tacp_ scope)* returns *ValidAfterRw*, then *np.tacp* is valid w.r.t. *eap.tacp_scope*. In light of these considerations, the *CheckEmergencyPolicy* function returns *ValidAfterRw*, when the rewritten emergency policy *np* satisfies all conditions (i-iv), i.e., when the administrative policy *eap* authorizes *np*.

Proof of Theorem 3.2.2
We have to prove that: 1) for each authorized emergency policy $rwep \in rwEPs$ returned by Algorithm 2, there exists at least an administration policy $eap \in EAPR$ such that *eap* authorizes *rwep*; 2) if $rwEPs = \emptyset$, then none of the administration policies $eap \in EAPR$ authorizes *ep* and *ep* cannot be rewritten in order to be valid w.r.t. at least an administration policy, i.e., *ep* is invalid w.r.t. any $eap \in EAPR$.
By Construction Algorithm 2 returns the emergency policy *ep* (line 5) passed as argument by a user u issuer of the policy (i.e., *ep* is validated) if $\exists$ $eap \in EAPR$ (lines 3-7) such that *CheckEmergencyPolicy(u, ep, eap)* (line 4) returns *Valid* (line 5). Lemma 3.2.3 ensures that if *CheckEmergencyPolicy(u, ep, eap)* returns *Valid*, then *ep* is valid w.r.t. to *eap*.
Algorithm 2 returns $\emptyset$, when *rwEPs* is empty (line 8). This becomes true (i.e., *ep* is not validated) if $\forall$ $eap \in EAPR$ (lines 3-7), *CheckEmergencyPolicy(u, ep, eap)* (line 4) returns *Invalid*. Lemma 3.2.3 ensures that if *CheckEmergencyPolicy(u, ep, eap)* returns *Invalid*, then *ep* is not valid w.r.t. *eap*. If this is true, for every $eap \in EAPR$, then *ep* is not valid w.r.t. any emergency administration policy.
Algorithm 2 returns a non-empty set *rwEPs* (line 8) of rewritten emergency policies if $\exists$ at least one $eap \in EAPR$ (lines 3-7) such that *CheckEmergencyPolicy(u, ep, eap)* (line 4) returns *ValidAfterRw* (line 6). Lemma 3.2.3 ensures that if *CheckEmergencyPolicy(u, ep, eap)* returns *ValidAfterRw*, then the rewritten emergency policy *np* is valid w.r.t. to *eap*. Since every rewritten emergency policy *np* is valid w.r.t. to *eap* and it is inserted into *rwEPs* (line 6), $\forall$ $rwep \in rwEPs$, then *rwep* is valid.

3.2.3 Emergency Policy Composition

The core emergency model is able to capture complex event patterns, but there exist critical scenarios that cannot be handled using this model. These are the cases of a combination of different emergency situations that may give rise to a new and more critical situation, requiring a new response plan, different from those plans already in place for the management of atomic emergencies. Therefore, we introduce the concept of *composed emergencies*, to describe how and which sub-emergencies have to be combined together to form a composed one.



```
FacilityMapsFilesPol
{
  sbj:FireFighters
  obj: MapsFiles
  priv: read
}

obl:
1. FireFightersCall
2. PoliceCall
```

```
SteamFilesPol {
  sbj:EPA Agents
  obj: SteamFiles
  priv: read
}

obl: FacilityEvacuation
```

```
ChemicalFilesPol {
  sbj:EPA Agents
  obj: ChemicalFiles
  priv: read
}

obl: WarnEPA
```

Figure 3.1: Industrial Facility Scenario: emergencies and related tacps/obligations

Example 3.2.11 *For example, consider the scenario of an industrial company facility which produces plastic material. Suppose that the facility is equipped with sensor networks detecting fire alarms, explosions and presence of toxic substances in air/water. Suppose that the system enforces the emergency policies represented in Figure 3.1: (1) when a fire alarm is detected, the firefighters and police agents are automatically called (i.e.,* FireFightersCall *and* PoliceCall *obligations) and they are allowed to access the facility map files (i.e.,* FacilityMapsFilesPol *temporary access control policy); (2) when an explosion emergency is detected, facility evacuation is enforced (i.e.,* FacilityEvacuation*) and the Environmental Protection Agency (EPA) personnel is allowed to read the files with information about steams processed in the facility (i.e.,* SteamFilesPol*); (3) when a high level of toxic substances is detected, the EPA should be warned about the emergency (i.e.,* WarnEPA*) and information on the chemical substances used in the facility should be immediately available to EPA staff (i.e.,* ChemicalFilesPol*). The situation might get much more critical if two or more of the above described emergencies are detected at the same time or in a sequence. Indeed, if the fire alarm is followed by an explosion, and the explosion is in turn followed by a toxic material loss* *emergency, it means that the fire and explosion caused damage with toxic material release. As such, the emergency situation requires the modification of the ongoing response activities. Since the risk of ecological disaster is high a higher level authority, such as the Department of Homeland Security (DHS), should be warned and any information about the processes executed in the facility should be immediately available to DHS staff.*

Situations like the one presented in Example 3.2.11 cannot be handled by emergency policies, since they require the new concepts of composed emergencies and related emergency policies, introduced in the following sections, in order to model new information sharing needs required by the composed emergency response plan.

We support two ways to specify how emergencies have to be combined together to from a composed emergency. The first represents the composed emergency as a list of multiple occurrences of the same emergency type. The second represents the composed emergency as a pattern of different emergencies. Formally, composed emergencies are defined as follows.

Definition 3.2.7 *A composed emergency ce is a pair (*combination, priority*), where* priority $\in \{high, low\}$ *indicates the priority of the composed emergency, whereas* combination *indicates the sub-emergencies and how they are combined together to from ce. More precisely, the* combination *component can be of one of the following forms:*

- $\{oc_1, ..., oc_n\}$*, such that* $oc_j = (emg_j, n_j)$*, where* emg_j *is an emergency identifier, whereas* $n_j \in \mathbb{N}$ *is the minimum number of* emg_j *instances necessary to trigger ce.*

- *pattern, which can be: (1) a* sequence emg_1, $emg_2[emg_1, s_2], \ldots , emg_n [emg_{n-1}, s_n]$*, where* emg_i *is an atomic or composed emergency, whereas* $emg_i[emg_{i-1}, s_i]$ *indicates that* emg_i *should happen between* emg_{i-1} *and a time interval of size* s_i*, defining in this way the sequence of emergencies* $emg_1, emg_2, ..., emg_n$*; (2) a* negation $\neg$ $emg[w]$*, which specifies the non-occurrence of emergency emg in a given time window w.*

Example 3.2.12 *Consider the situation presented in Example 3.2.11, where three atomic emergencies generate a critical situation, which can be modeled by the following composed emergency:*

$$EcologicalDisaster = (Pattern, high)$$

$$Pattern = \begin{cases} FireAlarm, \\ Explosion[FireAlarm, 1h], \\ ToxicMaterialLoss[Explosion, 1h] \end{cases}$$

If an explosion emergency is detected within one hour after the fire alarm emergency and a toxic material loss emergency is detected within one hour after the explosion, then the composed emergency EcologicalDisaster *is raised.*

The binding of an atomic or composed emergency with the corresponding tacps and obligations is modeled through emergency policies. In order to manage critical situations represented by composed emergencies, it is often necessary that the tacps and obligations that have been activated as response plan to sub-emergencies are overridden by tacps/obligations associated with composed emergencies.

For instance, consider again the scenario presented in Example 3.2.11, and suppose that the obligations associated with the sub-emergency fire alarm are: call firefighters and call police. When the composed emergency *EcologicalDisaster* is detected, while the police call should be deleted in order to not endanger the lives of police men with the released toxic material, the firefighters call should be maintained, since the fire must be extinguished regardless the release of toxic substances.

To handle situations like these, we enable the Emergency Manager to specify the overriding strategy determining how to behave with respect to the tacps/obligations associated with sub-emergencies involved in the composition. More precisely, we support three overriding strategies, that is, *maintain*, *delete*, and *block* that imply, respectively, that tacps/obligations associated with sub-emergencies are maintained, deleted or blocked until the end of the corresponding composed emergency.

The *block* overriding strategy is extremely important, since it allows temporary blocking tacps/obligations for the duration of the composed emergency. For instance, consider an emergency of power loss in a generator of a power plant and the consequent obligation *CallMaintenance*. If the emergency gets more critical, e.g., the generator is burning, a new obligation *FireFightersCall* should be enforced. In this case the *CallMaintenance* obligation should be blocked to not endanger maintenance staff lives, but it should be restated as soon as the fire is extinguished.

We are aware that there could be emergencies whose relevance requires not stopping any of the associated tacps/obligations, even in case these are involved in a composition. As an example, consider once again the scenario in Example 3.2.11, when the ecological disaster emergency is detected, although this composed emergency is more serious than its sub-emergency fire alarm, the related firefighter call should not be deleted/blocked until the fire is extinguished.

To prevent overriding of critical tacps/obligations, we introduce an *exception mechanism* at the emergency level. This is done by exploiting the priority level associated with each emergency (cfr. Definitions 3.1.14 and 3.2.7). More precisely, the tacps/obligations associated with any high priority emergency can be never deleted/blocked even though this emergency is involved in a composed emergency, and even though this latter requests for tacps/obligations overriding.[13] Moreover, considering that, given an emergency, not all of its response procedures have the same importance, we introduce a more fine-grained exception level, that is, the tacp/obligation level. More precisely, we enable the Emergency Manager to specify, for each emergency with low priority, those tacps/obligations that do not have to be deleted/blocked even in case the emergency is involved in a composed emergency which requests for tacps/obligations overriding. The formal definition of emergency policy is therefore modified as follows.

Definition 3.2.8 *An* Emergency Policy *is a tuple (*emg*,* tacps*,* obligations*,* overriding*), where* emg *is the identifier of an atomic or composed emergency;* tacps *is a set of pairs (*tacp*,*[14] exception*), where* tacp *is a temporary access control policy, whereas* exception $\in$ *{*true*,* false*};* obligations *is a set of pairs (*obl*,* exception*), where* obl *is an obligation, whereas* exception $\in$ *{*true*,* false*}. An exception will be used to denote whether a tacp or obligation enforces a policy/action that cannot be deleted or blocked (*exception = true*) by the overriding strategies. The* overriding *component consists of (*tacpOver*,* oblOver*), whose values in {*maintain*,* delete*,* block*} denote the overriding strategy for tacps/obligations, respectively.*

It is worth noting that in case *emg* is an atomic emergency, the *overriding* field is empty.

Example 3.2.13 *Consider the scenario in Example 3.2.11. The requirement to automatically warn the DHS, allowing its personnel to access information about any process in the facility (i.e., all files), when the* EcologicalDisaster *emergency (cfr. Example 3.2.11) is raised, can be modeled by associating the following policy to the* EcologicalDisaster *emergency, as shown in Figure 3.2.*

[13] In this report, we just consider two priority levels (i.e., high, low). We postpone as future work the definition of a more sophisticated exception mechanism.

[14] Tacps are expressed in terms of subject, object, privilege and context information.

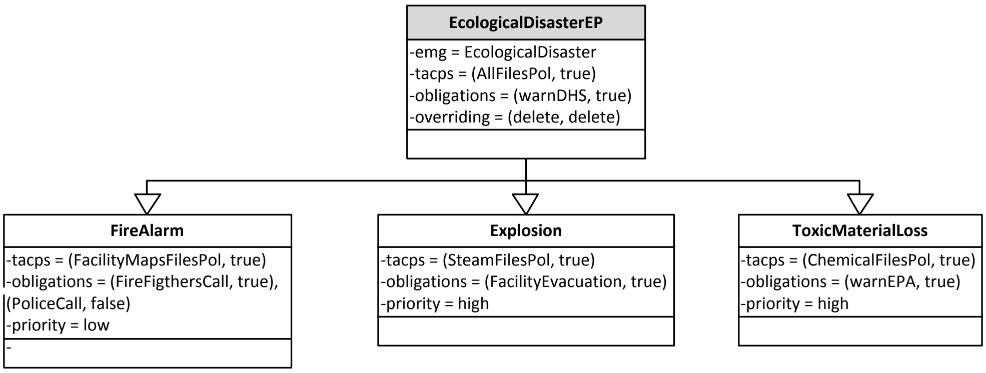

EcologicalDisasterEP
-emg = EcologicalDisaster
-tacps = (AllFilesPol, true)
-obligations = (warnDHS, true)
-overriding = (delete, delete)
FireAlarm
-tacps = (FacilityMapsFilesPol, true)
-obligations = (FireFigthersCall, true),
(PoliceCall, false)
-priority = low
-
Explosion
-tacps = (SteamFilesPol, true)
-obligations = (FacilityEvacuation, true)
-priority = high
ToxicMaterialLoss
-tacps = (ChemicalFilesPol, true)
-obligations = (warnEPA, true)
-priority = high

Figure 3.2: EcologicalDisaster Emergency Policy

In Figure 3.2, the EcologicalDisasterEP *emergency policy is represented, where* AllFilesPolicy *is a tacp granting DHS access to any file containing information about processes in the facility and* WarnDHS *obligation warns EPA about the ecological disaster emergency. Suppose now that the* FireAlarm *emergency has a low priority, whereas we assume that its tacp* FacilityMapsFilesPol*, allowing firefighters and police agents to access the facility maps files, has a true exception value. Moreover, two obligations are associated with* FireAlarm*, requiring to call firefighters and police agents, which we assume having a true and false exception value, respectively. Additionally, the* Explosion *emergency has a high priority and its tacp (called* SteamFilesPol*) and obligation (called* FacilityEvacuation*) have both true as exception value. Finally the* ToxicMaterialLoss *emergency has a high priority and its tacp (called* ChemicalFilesPol*) and obligation (called* WarnEPA*) have both true as exception value. Let us now explain the overriding strategies. Since tacpOver is set to delete, the tacp related to the low priority sub-emergencies (i.e.,* FireAlarm*) with exception field set to false (none in case of* FireAlarm*) are deleted, whereas those with a true exception value (i.e.,* FacilityMapsFilesPol*) are maintained. Similarly, since the flag oblOver is set to delete, obligations related to low priority sub-emergencies (i.e.,* FireAlarm*) with exception field set true (i.e., the obligation requiring to call firefighters) are maintained, whereas those with false exception value (i.e., the call police obligation) are deleted.*

Emergency Composition Tree

The introduction of composed emergencies brings new issues mainly related to overriding enforcement. Indeed, when a composed emergency ce is triggered, its sub-emergencies, say $e_1, \ldots, e_n$ have been already instantiated. This implies that the corresponding tacps/obligations have been already activated. If we further consider that each sub-emergency could be a composed emergency as well, the number of tacps/obligations linked to a composed emergency may be large. This may greatly impact the time needed to instantiate the new emergency e, since for each of the already inserted tacps/obligations it should be determined whether it has to be maintained, deleted or blocked. This decision is taken considering the overriding strategy associated with e, the priority of sub-emergencies as well as the exception values of the corresponding tacps/obligations. However, a key requirement for emergency management is to provide timely information to people involved in the response plan. A delay due to overriding enforcement could imply situations where information is not available, or available to the wrong people, due to tacps not yet overridden. Similarly, this delay can imply a delayed stop of risky activities imposed by obligations. To avoid these situations, we propose a solution where, for each composed emergency e for which a policy has been specified, the corresponding lists of tacps/obligations to be deleted, maintained or blocked are statically *pre-computed.* More precisely, we organize the tacps/obligations that have to be instantiated due to the triggering of an atomic/composed emergency as well as the lists of tacps/obligations to be overridden, into a set of tree data structures, called *Emergency Composition Tree.* In the following, we introduce the tree data structure and the algorithm for its generation.

An Emergency Composition Tree (*ECT*) is defined such that each emergency is represented as a node, whereas all information related to the corresponding policies are modeled as its attributes. The formal definition of an ECT is given in what follows.

Definition 3.2.9 *Given a composed emergency ce consisting of n subemergencies* $e_1, ..., e_n$ *and its corresponding emergency policy* cep*=(*ce*,* tacps*,* obligations*,* overriding*),*[15] *the corresponding ECT is defined as a pair* $\langle N, E\rangle$ *where:*

- $N = \{n_{ce},\, n_{e_1},\, \ldots,\, n_{e_n}\}$ *is the set of nodes. Node* n_{ce} *represents the composed emergency ce and has the following attributes:* tacps*,* obligations*,* priority*,* tacpOver*,* oblOver*,* tacpToDelete*,* tacpToBlock*,* oblTo Delete*, and* oblToBlock*. In particular,* tacps *and* obligations *contain the list of tacps/obligations specified in cep.tacps*[16] *and cep.obligations,*

[15] For simplicity, in Definition 3.2.9, we assume that each emergency is associated with a single policy. If an emergency is bound to multiple policies, Definition 3.2.9 can be easily extended.

[16] Here and in the following we use dot-notation to indicate fields of emergencies or polices (e.g., tacps, obligations).

priority *is the priority of emergency ce,* tacpOver *and* oblOver *represent the overriding strategies specified in cep.overriding,* tacpToDelete, tacpToBlock, oblToDelete *and* oblToBlock *contain, respectively, the tacps and obligations that have to be deleted or blocked in case of the triggering of ce. Each node* $n_{e_i} \in N$*,* $i \in [1, n]$ *represents a sub-emergency* e_i*. It has the same attributes as node* n_{ce}*, where* tacps *and* obligations *contain the list of tacps/obligations specified in the emergency policy related to* e_i*,* priority *is the priority of emergency* e_i*,* tacpOver *and* oblOver *represent the overriding strategies specified in the emergency policy associated with* e_i*, whereas the overriding lists (i.e.,* tacpToDelete, tacpToBlock, oblToDelete *and* oblToBlock*) contain tacps/obligations that have to be deleted or blocked in case of the triggering of* e_i*.*

- $E = \{(n_{ce}, n_{e_1}), \ldots, (n_{ce}, n_{e_n})\}$ *is the set of edges.*

Attributes related to overriding (i.e., *tacpOver*, *oblOver*, *tacpToDelete*, *tacpToBlock*, *oblToDelete* and *oblToBlock*) are optional. For instance, in case of a node denoting an atomic emergency they are unnecessary, as the following example clarifies.

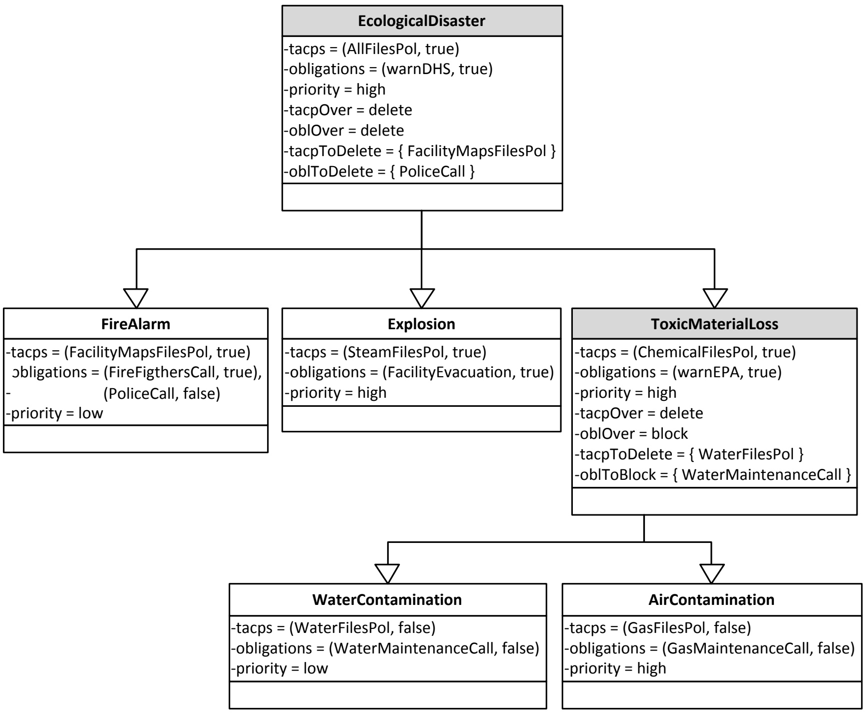

EcologicalDisaster
-tacps = (AllFilesPol, true)
-obligations = (warnDHS, true)
-priority = high
-tacpOver = delete
-oblOver = delete
-tacpToDelete = { FacilityMapsFilesPol }
-oblToDelete = { PoliceCall }
FireAlarm
-tacps = (FacilityMapsFilesPol, true)
ɔbligations = (FireFigthersCall, true),
- (PoliceCall, false)
-priority = low
Explosion
-tacps = (SteamFilesPol, true)
-obligations = (FacilityEvacuation, true)
-priority = high
ToxicMaterialLoss
-tacps = (ChemicalFilesPol, true)
-obligations = (warnEPA, true)
-priority = high
-tacpOver = delete
-oblOver = block
-tacpToDelete = { WaterFilesPol }
-oblToBlock = { WaterMaintenanceCall }
WaterContamination
-tacps = (WaterFilesPol, false)
-obligations = (WaterMaintenanceCall, false)
-priority = low
AirContamination
-tacps = (GasFilesPol, false)
-obligations = (GasMaintenanceCall, false)
-priority = high

Figure 3.3: Emergency Composition Tree

Example 3.2.14 *Consider the policy presented in Example 3.2.13 referring to the composed emergency* EcologicalDisaster *in Example 3.2.12. Suppose that* ToxicMaterialLoss *is defined as the composition of two atomic emergencies:* WaterContamination *and* AirContamination*. Suppose moreover that* WaterContamination *is a low priority emergency associated with the tacp* WaterFilesPol*, which allows water maintenance personnel to access files containing information about water usage in the facility. In contrast,* AirContamination *is a high priority emergency associated with the tacp* GasFilesPol *allowing the gas maintenance personnel to access files containing information about gas processed in the facility. The ECT corresponding to the* EcologicalDisaster *emergency is represented in Figure 3.3. The node associated with the composed emergency* ToxicMaterialLoss *has a* tacps *attribute which contains a tacp, called* ChemicalFilesPol*, with true exception value.* ChemicalFilesPol *allows EPA personnel to access files with information about chemical substances processed in the facility. The* obligations *attribute of* ToxicMaterialLoss *contains an obligation, called* warn EPA*, that warns EPA about the toxic material loss emergency. The* tacpOver *attribute is set to delete meaning that* ChemicalFilesPol *overrides the sub-emergency tacp* WaterFilesPol*, but not* GasFilesPol*. This is because* GasFilesPol *is associated with* AirContamination *which is a high priority emergency, whereas* WaterFilesPol *is associated with the low priority emergency* WaterContamination*. The* oblOver *attribute of* ToxicMaterialLoss *is set to block. However,* GasMaintenanceCall *has to still be enforced, since it is associated with the high priority emergency* AirContamination*, whereas the obligation* WaterMaintenanceCall *is temporary blocked until the end of* ToxicMaterialLoss*, since it is associated with the low priority emergency* WaterContamination*. Therefore, the overriding lists of* ToxicMaterialLoss *are the following:* tacpToDelete $=$ $\{$WaterFilesPol$\}$*,* tacpToBlock $= \emptyset$*,* oblToDelete $= \emptyset$*,* oblToBlock $=$ $\{$WaterMaintenanceCall$\}$*. The overriding lists of* EcologicalDisaster *are computed in a similar way.*

We show now how we create the set of ECTs for composed emergencies. Note that the same emergency could be part of one or more composed emergencies. To avoid storage of redundant information, we make use of an indexing data structure (i.e., a hash table), which maps each emergency with information about the position of the corresponding subtree in existing ECTs. The position is encoded as index$[emg] = (t_j, l_m, c_n)$, where t_j denotes an ECT, whereas l_m and c_n denote the position of the node related to emg in t_j (i.e., its level l_m and relative position c_n in the level, from left to right). Algorithm 3 receives as input the policy base CEP containing policies for composed emergencies and returns the set of created *ECTrees* and the associated indexing structure. For each policy $cep_j \in CEP$, it calls the *createECT()* function, which returns an ECT called *tree* and the modified *index* (line 3). *Tree* is then inserted into the *ECTrees* (line 4). Atomic emergencies that are not part of a composed emergency are not considered by Algorithm 3, therefore they are not indexed and during enforcement.

Algorithm 3: Emergency Composition Trees generation

```
  Input  : $CEP$ the composed emergency policy base
  Output: $ECTrees$, the set of ECTs, and the $index$ hash table
1 Let $index$ be an empty hash table, $root$ be an empty variable;
2 foreach $cep_j \in CEP$ do
3     <tree, index> = createECT ($cep_j$, root, 0, 0, index);
4     Insert tree Into ECTrees;
5 end
6 return  <ECTrees,index>
```

The $createECT()$ function receives as input: an emergency policy *cep* for a composed emergency; the *root* node of the ECT under construction (needed for the indexing); *depth* and *chNum*, denoting the level and number of children, where nodes created by the function will be inserted; the indexing hash table *index*. The $createECT()$ function returns a pair *<parent, index>* where *parent* is the node created for the input *cep* and *index* is the modified hash table. To better explain the meaning of *root*, *depth* and *chNum*, suppose to have a composed emergency ce_1 consisting in turn of two composed sub-emergencies ce_2 and ce_3. When the createECT function is called for ce_1 by Algorithm 3, *root* is an empty variable, *depth = 0* and *chNum = 0* (see line 3 in Algorithm 3), therefore *root* will be assigned to the node related to ce_1, say n_1 and emphindex[ce_1] = (n_1, 0, 0). When createECT is recursively called for ce_2, *root* = n_1, *depth = 1* and *chNum = 0*, therefore *index*[ce_2] = (n_1, 1, 0). Finally, when createECT is recursively called for ce_3, *root* = n_1, *depth = 1* and *chNum = 1*, therefore *index*[ce_2] = (n_1, 1, 1).

Function createECT calls function *createNode()*, by passing it *cep*, *root*, *depth*, *chNum* and *index* (line 1). This function returns a node, called *parent*, defined according to Definition 3.2.9, the modified *index* table and *root.* Then, function createECT analyzes each of the sub-emergencies sub_j involved in the input composed emergency (lines 4-14). If sub_j is an atomic emergency (line 6), then the function calls *createNode()*, which returns the *child* node (line 7), the modified *index* table and *root.* If sub_j is a composed emergency, then the *child* node is created calling recursively function *createECT()* (line 9). In both the cases, the *child* node is added as direct child of *parent* node (line 11). Finally, function createECT calls the *createOverLists()* function which inserts the proper tacps/obligations related to the *child* node into the overriding lists of the *parent* node (line 13). When all sub-emergencies have

Function createECT(cep, root, depth, chNum, index)

```
1  <parent, index, root> = createNode (cep, root, depth, chNum, index);
2  depth++;
3  Let $SubEmg$ be the set of sub-emergencies in cep.emg;
4  foreach $sub_j \in SubEmg$ do
5      Let $ep_j$ be the emergency policy associated with $sub_j$;
6      if $sub_j$ is an atomic emergency then
7          <child, index, root> = createNode ($ep_j$, root, depth, chNum, index);
8      else
9          <child, index> = createECT ($ep_j$, root, depth, chNum, index);
10     end
11     Create an edge between node parent and node child;
12     chNum++;
13     parent = createOverLists (parent, child);
14 end
15 return <parent, index>
```

been analyzed, then the createECT function returns *parent* node, i.e., the root of the created ECT and the modified *index* table (line 15).

The createNode function takes as input an emergency policy *ep*, the *root* of the ECT under construction, *depth* and *chNum*, denoting the level and number of children where the new node will be inserted. If a node associated with the emergency *ep.emg* already exists, then the createNode function gets that node through the *getNode* function (line 1), otherwise, a new node is created (line 3).

Function createNode(ep, root, depth, chNum, index)

```
1  if index[ep.emg] $\neq \emptyset$ then n = getNode(index[ep.emg]);
2  else
3      n = new node;
4      n.tacps = ep.tacps;
5      n.obligations = ep.obligations;
6      n.priority = ep.emg.priority;
7      if ep.emg is composed then
8          n.tacpOver = ep.tacpOver;
9          n.tacpOver = ep.oblOver;
10         n.tacpToDelete = n.tacpToBlock = n.oblToDelete = n.oblToBlock = $\emptyset$;
11     end
12 end
13 if depth = 0 then root = n;
14 Insert (root, depth, chNum) Into index[ep.emg];
15 return <n, index, root>;
```

The *tacps*, *obligations* and *priority* attributes are initialized for the new node (lines 4, 5, 6). If *ep.emg* is a composed emergency (line 7), then also the overriding strategies attributes are initialized (lines 8, 9) and the overriding lists are created (line 10). Finally, the createNode function creates the index (*root*,[17] *depth*, *chNum*) for node *n* (line 14) and returns the node itself, the modified *index* table and the *root* node (line 15).

Function createOverLists(parent, child)

```
1  if $child.priority \neq high \wedge parent.tacpOver \neq maintain$ then
2      foreach $tacp_j \in child.tacps$ do
3          if $tacp_j.exception \neq true$ then
4              if $parent.tacpOver = delete$ then
5                  Insert $tacp_j$ Into parent.tacpToDelete;
6              else
7                  Insert $tacp_j$ Into parent.tacpToBlock;
8      end
9  if $child.priority \neq high \wedge parent.oblOver \neq maintain$ then
10     foreach $obl_j \in child.obligations$ do
11         if $obl_j.exception \neq true$ then
12             if $parent.oblOver = delete$ then
13                 Insert $obl_j$ Into parent.oblToDelete;
14             else
15                 Insert $obl_j$ Into parent.oblToBlock;
16     end
17 return $parent;$
```

The createOverLists function takes as input a *parent* and a *child* node, inserts the proper tacps/obligations related to *child* into the overriding lists of *parent* and returns the modified *parent* node. If the *child* node is not linked with a high priority emergency and *parent* requests to override/block tacps (line 1), then the tacps related to *child* should be deleted or blocked. Therefore, each $tacp_j \in child.tacps$ is analyzed (lines 2-8) and if *exception* is not set to true (line 3) then: (i) if $parent.tacpOver = delete$ (line 4), $tacp_j$ is inserted into *parent.tacpToDelete* (line 5), (ii) if $parent.tacpOver = block$ (line 6), $tacp_j$ is inserted into *parent.tacpToBlock* (line 7). In lines 9-16, a similar overriding strategy is enforced for obligations.

Policy Enforcement

Emergency policy enforcement is done by making use of ECTs. More precisely, when an emergency *e* is detected, then: if *e* is an atomic emergency, its tacps and obligations are inserted into the system, whereas if it is composed, tacps and obligations contained into *tacps* and *obligations* attributes

[17] The *root* variable is already set if *depth* is greater than zero, but in case *depth* is equal to zero it means that node *n* is the root of its ECT, thus variable *root* is set to *n* (line 13).

of the corresponding ECT node are inserted into the system and those contained in *tacpToDelete*/ *oblToDelete*, *tacpToBlock*/*oblToBlock* are deleted and/or blocked, respectively.

Example 3.2.15 *Consider the ECT in Figure 3.3. When the composed emergency* EcologicalDisaster *is detected, active tacps are:* FacilityMapsFilesPol *(linked to* FireAlarm*),* SteamFilesPol *(linked to* Explosion*),* GasFilesPol *(linked to* AirContamination*) and* ChemicalFilesPol *(linked to the* ToxicMaterialLoss*). In contrast, the* WaterFilesPol *tacp was already overridden by the* ChemicalFilesPol *tacp. The node related to* EcologicalDisaster *is retrieved using the indexing data structure. Then, the system enforces tacps and obligations related to* EcologicalDisaster*. The system also retrieves the tacps contained in the overriding lists. Since* tacpToDelete *= {* FacilityMapsFilesPol *},* FacilityMapsFilesPol *is deleted. The obligations in place in the system are:* FireFightersCall *and* PoliceCall *(linked to* FireAlarm *emergency),* FacilityEvacuation *(linked to* Explosion*),* GasMaintenanceCall *(linked to* AirContamination*), and* warn EPA *(linked to the* ToxicMaterialLoss*), whereas the* WaterMaintenanceCall *obligation was already blocked until the end of the* ToxicMaterialLoss *emergency. The system checks the obligations contained in the list associated with the* EcologicalDisaster *node. Since* oblToDelete *= {* PoliceCall *},* PoliceCall *is deleted.*

Complexity Analysis

In this section, we estimate the time needed to create the set of ECTs, which are generated at policy specification time, as well as the time needed for composed emergency policy enforcement using the generated ECTs.

ECTs generation To estimate the time required to create the set of ECTs, we analyze Algorithm 3. We first analyze createECT function, then we draw conclusions about complexity of Algorithm 3.

Function createECT: As a first step, createECT calls createNode by passing as input *cep*, a policy for composed emergencies (line 1). The time required by all operations in the createNode function is a constant time c except the time to copy the list of tacps and obligations (lines 4, 5) which is linear in the number of tacps n_t and the number of obligations n_o. Therefore the total time required by the function is $n_t + n_o + c$, i.e., $O(n_t + n_o)$. Then, the createECT function considers each sub-emergency sub_j contained into *cep* (lines 4-14). When sub_j is an atomic emergency, the function creates the corresponding node, by calling function createNode (line 7), whereas if

sub_j is a composed emergency, it recursively calls itself (line 9). For each sub-emergency the createECT function calls also the *createOverLists* function (line 13) which implements the overriding strategy. The time required by this function depends again on the number of tacps and obligations in *child* node, i.e., $O(n_t + n_o)$. To give an estimation of the total time required by createECT function, we assume that the number of sub-emergencies that are involved at any level in *cep* is n and all sub-emergencies are composed,[18] which means that createECT is recursively called n times. The overall time is, in the worst case, $O(n \times (max(n_t) + max(n_o)))$ where $max(n_t)$ and $max(n_o)$ denote the maximum number of tacps/obligations associated with policies of all sub-emergencies. Therefore, the overall time is linear in the number of sub-emergencies.

Main Algorithm: Algorithm 3 calls function createECT for each policy associated with a composed emergency (lines 2-5). Let m be the number of emergency policies associated with composed emergencies. Then, the overall time required for Algorithm 3 is: $O(m \times n \times (max(n_t) + max(n_o)))$. Thus, Algorithm 3 is linear in the number of emergency policies for composed emergency and, since the creation of each ECT takes a linear time in the number of sub-emergency, Algorithm 3 is efficient and scalable.

Emergency Policy Enforcement Analysis

Thanks to the proposed tree and indexing data structures, composed emergency enforcement is efficient in terms of time needed to decide which tacps/obligations have to be inserted, deleted or blocked. We recall that, for a policy associated with a composed emergency *ce*, the enforcement consists of the following stpdf: (i) retrieval of the ECT node related to the emergency, (ii) reading of the tacps and obligations attributes and (iii) insertion in the policy bases of the retrieved tacps/obligations, (iv) reading of the overriding lists, and (v) execution of the overriding operations (i.e., delete/block the overridden tacps/obligations). By using the defined data structures, the time needed to perform step (i) is expected to be short. Indeed, given a composed emergency *ce*, retrieving the root node of the corresponding subtree in an ECT requires just to access the first entry in the hash table associated with *ce*, which requires a constant small time. Once index (t, l, c) has been retrieved from the hash table, the time needed to access the indexed node is again very small, as it requires to access node at level l and internal position

[18] Actually, at least one emergency among those involved in the composition has to be atomic, but to estimate the worst case, we are assuming that they are all composed as this requires more time.

c, i.e., the complexity is $O(l * c)$. Stpdf (ii) and (iii) require reading two node attributes (i.e., *tacps* and *obligations*) and inserting their content into the proper repository. Assuming that read and write operations require a constant time, then these stpdf have a time complexity of $O(n_t + n_o)$, where n_t is the number of tacps and n_o is the number of obligations. The time required by both stpdf (iv) and (v) is linear in the lists size (i.e., the number of items to be read and written). As such, let $max(n_l)$ be the maximum size of the overriding lists, the overall complexity is $O(max(n_l))$. Therefore, the overall cost of policy enforcement is $O(l * c + n_t + n_o + max(n_l))$.

Chapter 4

Unspecified Emergency Management

The *core emergency policy* model proposed and its extensions (i.e., composed emergency policies and administration policies) are able to cover emergencies which can be specified *a priori*. However, there are many scenarios where this might not be enough, since there are emergencies which cannot be predicted beforehand. For instance, in healthcare domain it is difficult to specify in advance any possible injury or disease which might be considered as an emergency. These unspecified emergencies are not detected by the system and they are not connected to policies allowing information sharing needs (unspecified policies). This may have serious consequences, for instance in healthcare domain this might endanger human lives.

Consider, for example, a hospital where patients wear sensors for real time monitoring of their vital signs; usually only the doctor in charge of a patient is allowed to read his/her medical record. Suppose that during the night the ECG wave of a patient p shows an interference dissociation (i.e., a rare case of arrhythmia, which might have not been modeled as an emergency) and his/her doctor d_1 is not working at the time of the emergency, but another doctor d_2 requires to access the medical record of p. In this case, we might have that: (1) arrhythmia emergency and related policy have been specified, thus the emergency is detected and d_2 is allowed to access the medical record of p due to an existing emergency policy; (2) arrhythmia emergency and related policy are unspecified, thus the emergency is not detected and d_2 is not allowed to access the medical record of p. We might agree that in case (2) not allowing d_2 to access the medical record might endanger patient p life.

The basic idea is to detect unspecified emergencies exploiting anomaly detection techniques and to permit those access requests that should be denied due to the absence of policies related to unspecified emergencies. For instance, in case (2), the arrhythmia emergency can be easily detected as an anomaly, i.e., during arrhythmia values of heart rate have not a regular frequency and the consequent access request by d_2 should be permitted because it is related to the emergency, i.e., they refer to the same patient.

Obviously, not all access request related to an emergency should be allowed, but only those access requests related to unspecified emergencies. In order to detect whether an access request is related to an unspecified emergency or it is an attempted abuse, we determine if the access request is close to satisfy existing policies. We call this technique *policy based analysis*. For instance, in case (2), the access request performed by d_2 is very similar to the regular policy for d_1 since they are both doctors and the protected object is the same, i.e., the medical record of patient p.

The management of unspecified emergencies is based on three strategies: the *policy based analysis*, *anomaly based analysis* and *historical based analysis*. The *policy based analysis* calculates how much an access request is close to satisfy existing policies. The anomaly based analysis combines anomaly detection techniques and complex event processing (CEP) in order to detect anomalous events which might represent unspecified emergencies and correlate these events to denied access requests. The historical based analysis considers previously permitted access requests in order to detect if the current access request is similar to one of them. For each of these strategies, we define measures called *satisfaction level*, *anomaly level* and *historical level*. These levels measure, respectively, how much an access request is close to satisfy existing policies, how much a set of events is anomalous w.r.t. the normal behavior, how much an access request is similar to the previously permitted access requests. Moreover, we present a large set of experiments which show not only the effectiveness of our strategies, but also how to combine our measures in order to maximize the detection of unspecified emergencies.

4.1 Detection and Management of Unspecified Emergencies

In this section, we introduce our framework to extend the model presented in [27] in order to deal with unspecified emergencies. As we claimed in the introduction, a denied access request should be authorized if it represents an information need for an unspecified emergency. Unspecified emergencies are detected as anomalies, which are patterns in data that do not conform to expected normal behavior [33]. For example, suppose the average heart rate of a patient is around 80 bpm (i.e., the normal behavior), if a pattern of events with heart rate of 120 bpm has been detected, it represents an anomaly. In order to detect and manage anomalous events which might represent unspecified emergencies, we have identified three characteristics to discover whether an access request expresses an information need for an unspecified emergency or not: *the access request is (i) close to satisfy existing policies*, *(ii) related to an anomaly*, *(iii) similar to previously authorized access requests*.

If an access request is *close to satisfy existing policies (i)*, it is likely that the access request represents an information need for an unspecified emergency, otherwise it might represent an attempted abuse. For instance, suppose to have a policy that authorizes doctors to read patients medical records and suppose that a user with another role has performed an access request for a patient medical record; if the role of this user is close to the doctor role (e.g., paramedic), the access request might represent an information need for an unspecified emergency. On the contrary, if the role of this user is very different from the doctor role (e.g., administrative staff), the access request might represent an attempted abuse.

If an access request is *related to an anomaly (ii)*, it is likely that the anomaly represents an unspecified emergency and the access request represents the related information need necessary to manage the emergency situation. For example, consider the pattern of anomalous hear rates of 120 bpm introduced before and suppose these events belong to patient p, moreover, suppose a paramedic requires to read p medical record and the anomalous heart rate values has been detected two minutes before the access request; it is likely that patient p is under an unspecified emergency situation and the paramedic needs to read his/her medical record to give first aid.

If an access request is *similar to previously authorized access requests (iii)*, it is likely that the current access request should be authorized since similar ones have been authorized before. For instance, suppose a paramedic p_1 has been authorized to read patient p_1 medical record and another paramedic p_2 is trying to read patient p_2 medical record, it is likely that the two paramedics both need to read p_1 and p_2 medical records in order to manage unspecified emergencies.

Based on the three dimensions described above, the detection of unspecified emergencies is performed using three different strategies: (1) *policy based analysis* (2) *anomaly based analysis*, and (3) *historical based analysis*. The architecture of our framework is shown in Figure 4.1.

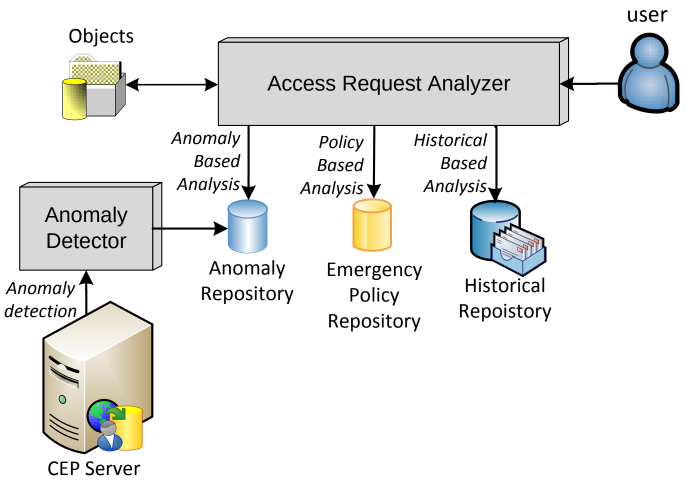


Figure 4.1: Framework Architecture

The architecture presented in Figure 4.1 ensures an effective management of unspecified emergencies and related information sharing requirements exploiting the following three strategies.

- **Policy Based Analysis:** every time an access request is denied, the *Access Request Analyzer* compares it against temporary access control policies in the *Emergency Policy Repository* in order to find the tacp which is the closest to be satisfied by the access request. This policy based analysis returns a score called *satisfaction level* which represents how much the access request is close to satisfy the tacp.

- **Anomaly Based Analysis:** this strategy is divided into two parts: (i) *anomaly detection* and *anomaly correlation*. The first part is performed by the *Anomaly Detector* which monitors events coming from the CEP system in order to detect anomalous events which might represent unspecified emergencies. Once a set of anomalous events is detected, an anomaly is stored along with an *anomaly level* (i.e., how much events are anomalous w.r.t. the normal behavior) in the *Anomaly Repository*. The *anomaly correlation* is performed every time an access request is denied. In this phase, the *Access Request Analyzer* searches, in the *Anomaly Repository*, anomalies occurred in a specified time window before the access request in order to find the anomaly which is the most correlated to the access request. The anomaly based analysis returns the *anomaly level* of the most correlated anomaly.

- **Historical Based Analysis:** every time an access request *ar* is denied, the system finds the related anomaly *an*, if exists, exploiting our anomaly based analysis techniques. Then, *ar* and *an* are compared against previously permitted access requests in order to find the couple (ar_h, an_h) which is the most similar to *ar* and *an* in both access request and anomalous events, if exist. The historical based analysis returns a score called *historical level* which represents how much *ar* and *an* are similar to the couple (ar_h, an_h).

Once these three strategies has been performed the anomaly level of the anomaly related to the denied access request, the satisfaction and historical level of access request are combined together to give an overall level *ol* which is used to decide to authorize or not the access request. More precisely, the *Access Request Analyzer* compares *ol* against a predefined threshold *th* and a tolerance value ε and if $ol > th + \varepsilon$, the access request is authorized, if $ol < th - \varepsilon$, the access request is denied, otherwise the access request is considered ambiguous. In case, the access request is authorized, it is stored in the *Historical Repository* with the related anomaly, if exists.

In the following sections, we give a detailed explanation of the *policy based analysis*, *anomaly based analysis* and *historical based analysis.*

4.2 Policy Based Analysis

Before introducing *policy based analysis*, it is important to give a formal specification of two building blocks of our model such as event type and object type. An event type `et` and an object type `ot` specify the schema of events belonging to `et` and the schema of objects belonging to `ot`, respectively. Their formal definitions are provided in the following.

Definition 4.2.1 ***(Event Type):*** *An event type `et` is a couple (`schema`, `identifier`), where `schema` is a set of couples (`att`, `type`) where `att` is the name of an attribute whereas `type` is its data type, `identifier` is the name of an attribute belonging to `schema`*[1]*.*

This definition is slightly different from the one proposed in Section 3.1 since the event type is not just the result of a query over a data stream, but it contains the schema of the data stream events and an identifier which is an attribute that uniquely identify group of events, e.g., patient id = 1, uniquely identify events associated with patient 1. In light of these modification, it is

[1] The role of the identifier is clarified in Example 4.2.1

useless to specify an identifier in the emergency specification, since the emergency takes the same identifier of the event type over which the emergency is defined (this is clearer in Example 4.2.1).

Definition 4.2.2 ***(Object Type):*** *An object type* `ot` *is a set of couples* `(att, type)` *where* `att` *is the name of an attribute whereas* `type` *is its data type.*

The following example clarifies the event and object types and present an emergency and related tacp specifications which are used to explain policy based analysis.

Example 4.2.1 *Consider the healthcare scenario where patients are constantly supervised through a specialized equipment which ensure a real time monitoring of their vital signs. Data gathered by the monitoring equipment is sent to the CEP in order to automatically detect emergency situations. More precisely, we suppose that each sensor sends patients vital signs to the CEP through the following event type.*

```
VitalSigns{
  schema = {(heart_rate, int), (glucose_level, int),
             (patient_id, string)}
  identifier = patient_id;
}
```

The event type `VitalSigns` *represents events containing 3 attributes: two numerical integer values called* `heart_rate` *and* `glucose_level` *and a string value called* `patient_id`*. The identifier attribute is* `patient_id` *whose role is crucial in emergency specification as shown in the following. In this scenario, a hyperglycemia emergency can be defined over the* `VitalSigns` *event type as follows.*

```
HyperglycemiaEmergency {
    init: VS1 $v_1$;
    VS1 = σ(glucose_level > 200)(VitalSigns);
    end: VS2 $v_2$;
    VS2 = σ(glucose_level ≤ 200)(VitalSigns);
    timeout: ∞;
}
```

The emergency starts when the glucose level in the blood of a patient is higher than 200 mg/dl and it ends when the glucose level of the same patient returns lower than or equal to 200 mg/dl. The matching between the `init` *and* `end` *event is ensured by the identifier* `patient_id` *defined in* *the* `VitalSigns` *event type. During hyperglycemia emergency paramedics taking care of the patient under emergency may be authorized to read objects belonging to the object type* `MedicalRecord` *which is defined as follows.*

```
MedicalRecord {(patient, string), (name, string),
               (lastname, string), (age, int)}
```

The object type `MedicalRecord` *may be used in the following tacp to authorize user belonging to the role doctor and working in the intensive care ward to read the medical record of patients during hyperglycemia emergency.*

```
HyperglycemiaPolicy (hgp) {
  s_hgp = {
    sr_hgp = doctor
    sc_hgp = (ward = Intensive Care Ward)
  }
  o_hgp = {
    ot_hgp = MedicalRecord
  }
}
```

The *policy based analysis* measures how much a denied access request is close to satisfy existing temporary access control policies. Once an access request is denied, the system stores this decision together with additional information into a structure called *denied access request (dar)* whose formal definition is the following.

Definition 4.2.3 ***(Denied Access Request):*** *A denied access request dar is a tuple (s_{id}, o_{id}, p_d, s_d, o_d), where s_{id} identifies the user to which has been denied the exercise of privilege p_d on the target object o_{id}; s_d = (SR_d, SA_d) contains information related to s_{id}, where SR_d is the set of roles assigned to him/her, and SA_d is his/her profile, defined as a set of attributes and corresponding values (a, v) of s_{id} profile; o_d = (ot_d, OA_d), where ot_d denotes the object type of o_{id} and OA_d is a set of attribute and related values (a, v) associated with object o_{id}.*

Example 4.2.2 *Consider the healthcare scenario presented in Example 4.2.1 and the tacp* `HyperglycemiaPolicy (hgp)`. *Let us assume that an access request has been denied, thus the following dar is stored into the system.*

```
dar1 {
  s_id = paramedic1
  o_id = MedialRecord1
  s_dar1 = {
    $SR_{dar_1}$ = {paramedic}
    $SA_{dar_1}$ = {(ward, Cardiac Ward)}
  }
  $o_{dar_1}$ = {
    $ot_{dar_1}$ = MedicalRecord
    $OA_{dar_1}$ = {(patientid, 1)}
}
}
```

`Dar1` represents an access request performed by the user `paramedic1` belonging to the paramedic *role working in the* Cardiac Ward *and requesting to access `MedialRecord1` belonging to patient 1.*

We need a measure for estimating the *satisfaction level* of a given dar d w.r.t. a tacp t. We recall that a tacp states (i) the subject specification in terms of conditions on user profile attributes and roles, and (ii) the object specification in terms of conditions on object type and properties. Thus, we are interested in measuring: how much the profile and roles of the user who has submitted d is far to satisfy conditions in the subject specification in t, and, how much the object requested in d is close to satisfy the conditions on the object description in t. We refer to these measures as *satisfaction level of d on t w.r.t. the subject specification*, and *w.r.t. the object specification.* The overall satisfaction level of d on t is then defined as combination of these two measures. Note that, object and subject specifications have some common features. Indeed, they both state constraints on *categories*, that is, roles and object types in the subject and object specifications, respectively. Moreover, both of them specify *conditions on attributes* of user profiles and object properties. Thus, to estimate both satisfaction levels a similar process has to be performed. This implies to measure (1) how the requested category (i.e., role, object type) is far from the ones of user/object (i.e., roles assigned to requesting user, types of requested object) and (2) how a given set of attribute values (i.e., values of user profile and values of object properties) is far to satisfy conditions over them (i.e., conditions on user profile, conditions on object properties). For this reason, in the following we first introduce how we estimate (1) and (2).

4.2.1 Satisfaction level for roles and object types

In order to measure these satisfaction levels, we have to estimate how two roles/object types are close each other. At this aim, having objects and roles organized in hierarchies helps, in that we can exploit existing distance measure defined for hierarchies [64]. More precisely, we assume to have a distance function $d_H(v_1, v_2)$ which takes as input two values v_1 and v_2 (i.e., two roles, two object types) and returns a value in the range $[0, 1]$ as a measure of how much v_1 is close to v_2 in a given role/object type hierarchy H. It is important to note that, even though roles and object types are not hierarchically organized, it is possible to calculate $d_H()$ distance using, instead of hierarchy H, a generic dictionary based ontology such as WordNet [78]. In order to keep the presentation as general as possible, we do not define the distance function d_H, but, where needed, i.e., examples and experiments, we adopt the Wu and Palmer measure [79]

Definition 4.2.4 ***(role and object satisfaction level):*** *Let t be a tacp, where sr_t and ot_t denote the role and object type specified in subject and object specification, respectively. Let d be a dar, where SR_d and ot_d denote the set of roles assigned to user specified in d and the object type of requested object, respectively. Let H_r, H_o be the hierarchies for roles and object types, respectively. The satisfaction level of d on t w.r.t. roles, denoted as $rsl(t, d)$, and the satisfaction level of d on t w.r.t. object types, denoted as $osl(t, d)$, are defined as follows:*

$$rsl(t, d) = 1 - min(d_{H_r}(sr_t, sr_{d_1}), \ldots, d_{H_r}(sr_t, sr_{d_n}))\forall\ sr_{d_j} \in SR_d \tag{4.1}$$

$$otsl(t, d) = 1 - d_{H_o}(ot_t, ot_d) \tag{4.2}$$

The role satisfaction level is defined as one minus the minimum distance between each role assigned to the dar user and the role specified in the tacp, whereas the object satisfaction level is specified as one minus the distance between the object type requested in the dar and the object type specified in the tacp.

4.2.2 Satisfaction level for subject/object conditions

Given a tacp and a dar, we have to measure: (i) how much the user profile in the dar is close to satisfy the Boolean expression in the tacp subject specification, and (ii) how much properties of the target object are close to satisfy the Boolean expression in the tacp object specification. In both cases, we need a measure to state how far a given set of attribute values is to satisfy conditions posed on the corresponding attributes. For example, a subject specification, where we have a condition *"ranking > 5"*, and two users u_1 and u_2 whose profiles have *ranking = 5* and *ranking = 1*. None of them satisfy the Boolean expression, but we can say that u_1 is closer to satisfy it than u_2. Similar examples hold for conditions on object properties. At this purpose, we convert the Boolean expression in a Disjunctive Normal Form (DNF) to calculate the satisfaction level in a bottom up way. We first evaluate satisfaction levels between user profile/object properties and each predicate, then we use these values to estimate satisfaction level of each clause, up to the entire expression.

Given a predicate p and an attribute value v_d (i.e., value of a profile attribute, value of an object property), its satisfaction level is calculated in a different way based on type of the attribute in p. If it is a *numerical attribute*, this is calculated as the normalized Euclidean distance between v_d and v_p,[2], whereas if it is *categorical attribute* it is calculated as the distance $d_H()$ between v_d and v_p in the hierarchy over which the attribute is organized.[3]

Definition 4.2.5 ***(satisfaction level of a predicate):*** *Let $p = a_p\ \theta\ v_p$ be a predicate, where a_p is an attribute name,[4] $\theta \in \{<, >, =, \leq, \geq\}$, and v_p is a constant value. Let A be a set of pairs (a_i, v_i) denoting attribute names and corresponding value. The predicate satisfaction level of p on A, is calculated by the psl function.*

$$psl(p, A) = \begin{cases} 0 & \textit{if}\ \nexists\ (a_i, v_i) \in A \mid a_i = a_p \\ 1 & \textit{if}\ \exists\ (a_i, v_i) \in A \mid a_i = a_p \\ & \wedge\ v_i \in sat(p) \\ 1 - d_H(v_p, v_i) & \textit{if}\ \exists\ (a_i, v_i) \in A \mid a_i = a_p \\ & \wedge\ v_i \notin sat(p) \\ & \wedge\ a_p\ \textit{is categorical} \\ 1 - \frac{|v_p - v_i|}{|Dom(a_p)|} & \textit{otherwise} \end{cases} \tag{4.3}$$

where $Dom(a_i)$ is the domain of attribute a_i and $sat(p)$ is the set of values which satisfy predicate p.
Let d be a dar, where SA_d and OA_d denote the set of profile attributes and object properties, respectively, the satisfaction level of p on d w.r.t. subject specification and w.r.t. object specification are defined as $psl(p, SA_d)$ and $psl(p, OA_d)$, respectively.

[2] Note that this does not apply to those attributes defined as identifiers, since measuring the distance between two identifiers is meaningless.

[3] As explained for roles/object types, even though an attribute does not belong to a domain specific hierarchy it is always possible to calculate the distance d_H using a generic dictionary based ontology such as WordNet [78].

[4] We assume there are no syntactic variations for attribute names. For instance, it is not possible to have attribute names such as "rank" and "ranking" in different tacps referring to the rank attribute. A schema matching approach [21, 46, 40] can be adopted to unify attribute names before measuring predicates satisfaction levels.

According to Equation 4.3, the predicate satisfaction level is equal to: (i) zero, in case the attribute a_p is not included in the attributes of the user profile/object properties; (ii) one, in case the attribute a_i is included in the attributes of the user profile/object properties and the attribute value v_i satisfies the predicate; (iii) one minus the d_H distance between v_p and v_i in a given hierarchy H, in case the categorical attribute a_i is included in the attributes of the dar subject specification, but its value v_i does not satisfy the predicate p; (iv) one minus the normalized Euclidean distance between v_p and v_i, otherwise.

Based on above definition we can define satisfaction level of a clause as follows.

Definition 4.2.6 ***(satisfaction level of a clause):*** *Let $c = p_1 \wedge \cdots \wedge p_n$ be a clause, A be a set of pairs (a_i, v_i) denoting attribute names and corresponding values. The satisfaction level of c on A is calculated by the clsl function.*

$$clsl(c, A) = \frac{psl(p_1, A) + \cdots + psl(p_n, A)}{n} \tag{4.4}$$

Let d be a dar, where SA_d and OA_d denote the set of profile attributes and object properties, respectively, the satisfaction level of clause c on d w.r.t. subject specification and w.r.t. object specification are defined as $clsl(p, SA_d)$ and $clsl(p, OA_d)$, respectively.

Finally, since each clause c is part of a disjunctive clause (i.e., the DNF of the subject condition or object conditions), the satisfaction level of DNF on a set of attribute A is computed as the maximum value of clause satisfaction levels between each conjunctive clause c_i and attributes in A.

Definition 4.2.7 ***(satisfaction level of DNF):*** *Let $dnf_t = c_1 \vee ... \vee c_n$ be the DNF of a subject or object condition in a tacp t, and A be a set of pairs (a_i, v_i) denoting attribute names and corresponding values. The satisfaction level of dnf_t on A is calculated by the $dnfsl$ function.*

$$dnfsl(dnf_t, A) = max(clsl(c_1, A), \ldots, clsl(c_n, A)) \tag{4.5}$$

Let d be a dar, where SA_d and OA_d denote the set of profile attributes and object properties, respectively, the satisfaction levels of the DNF dnf on d w.r.t. subject specification and w.r.t. object specification are defined as $dnfsl(dnf_{sc_t}, SA_d)$ and $dnfsl(dnf_{oc_t}, OA_d)$, respectively.

4.2.3 Satisfaction Level of a dar on a tacp

Given a tacp t and a dar d, we can estimate the *satisfaction level of d on t w.r.t. the subject specification* and *w.r.t. the object specification* exploiting the above definitions.

Definition 4.2.8 ***(satisfaction level of d on t w.r.t. the subject specification and w.r.t. object specification):*** *Let t be a tacp, and (sr_t, sc_t), (ot_t, oc_t) be its subject and object specifications, respectively, where sc_t and oc_t are expressed in DNF. Let d be a dar, where (SR_d, SA_d), (ot_d, OA_d) denote roles and profile of requesting user, types and properties of requested object, respectively. The satisfaction level of d on t w.r.t. the subject specification is defined as follows:*

$$sbjsl(s_t, s_d) = \frac{w_1 * rsl(sr_t, SR_d) + w_2 * dnfsl(sc_t, SA_d)}{2} \tag{4.6}$$

The satisfaction level of d on t w.r.t. the object specification is defined as follows:

$$objsl(o_t, o_d) = \frac{w_1 * osl(ot_t, ot_d) + w_2 * dnfsl(oc_t, OA_d)}{2} \tag{4.7}$$

Both equations calculate the satisfaction levels as the average of the DNF satisfaction levels computed on attributes (Definition 4.2.7) and satisfaction level on roles/object types (Definition 4.2.4). The weights values w_1 and w_2 can be used for emphasizing the importance of the role/object or condition satisfaction levels, respectively.

Finally, we can define *satisfaction level of d on t* by combing the above levels.

Definition 4.2.9 ***(Satisfaction level of a dar on tacp):*** *Let t be a tacp, where s_t, o_t denote its subject and object specifications. Let d be a dar, where s_d, o_d denote description of requesting user and requested object (see Definition 4.2.3). Satisfaction level of d on t is calculated by the* t-darsl *function:*

$$t - darsl(t, d) = \frac{sbjsl(s_t, s_d) + objsl(o_t, o_d)}{2} \tag{4.8}$$

Equation 4.8 calculates tacp-dar satisfaction level as the average between the subject satisfaction level $sbjsl$ and the object satisfaction level $objsl$ (Definition 4.2.8).

Example 4.2.3 *Consider the denied access request* `dar1` *presented in Example 4.2.2. In this example, we suppose roles, object types and attributes are hierarchically organized as shown in Figure 4.2. In this case, the role and object satisfaction levels are calculated as follows: (1)* $rsl(hgp, dar1) = 1 - min(d_{H_r}(doctor,$ $paramedic)) =$ *1 - 0.33*[5] *= 0.67, where the distance value is small, i.e., 0.33, since the two roles are very close in the hierarchy; (2)* $otsl(hgp, dar1) = 1$, *since the two objects have the same type. The subject condition in hgp contains only one predicate* p:(ward = Intensive Care Ward) *and the set of subject attributes in* $dar1$ *is* $SA_{dar_1} = \{$(ward, Cardiac Ward)$\}$. *Since ward is a hierarchically organized attribute and cardiac ward does not satisfy the predicate p, the predicate satisfaction level is calculated as follows:* $psl(p, SA_{dar_1}) = 1 - d_H($Intensive Care Ward, Cardiac Ward*)= 1 - 0.33 = 0.67.*

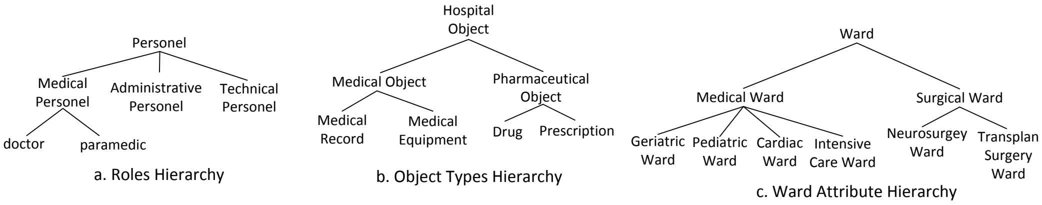


Figure 4.2: Roles, Object Types and Ward Attribute Hierarchies

Since the subject condition is composed of one predicate $dnfsl(sc_{hgp}, SA_{dar1}) = 0.67$. *Combining these values, it is possible to calculate subject and object satisfaction levels as follows: (1)* $sbjsl(s_{hgp}, s_{dar1}) = \frac{0.67+0.67}{2} = 0.67$*; (2) since the object specification contains only the object type and the two object have the same type* $objsl(o_{hgp}, o_{dar1}) = 1$. *The overall tacp-dar satisfaction level is calculated as the average of these two values:* $t - darsl(hgp, dar1) = \frac{0.67+1}{2} = 0.835$. *This level is returned as the result of the* policy based analysis *phase.*

The *t-darsl* is used to find the tacp which is the closest to be satisfied by the current denied access request, i.e., the tacp with the highest satisfaction level. Obviously, it is inefficient to compare a dar against each tacp, especially if the number of tacps is large. In order to avoid such sequential comparison, we make use of a pre-computed *roles-objects matrix*. This matrix allows quickly selecting a subset of tacps whose roles and object type specifications are close to be satisfied by the dar. Then, among the selected policies the

[5] In this Example the distance between the two roles in the hierarchy is calculated using the Wu and Palmer measure [79].

tacp-dar satisfaction level is calculated and the maximum satisfaction level is return as result of the policy based analysis.

More precisely, let R be the set of roles defined in the entire access control system and O be the set of object types. The roles-objects matrix is a matrix with a row for each role $r_i \in R$ and a column for each object type $o_j \in O$. A cell in the row r_i and column o_j contains a list of tacps order by their satisfaction level. Please note that we cannot use *t-darsl* measure to order the tacps in each matrix cell, since we need to measures how much a role r_i is close to satisfy a tacp subject specification and how much an object o_j is close to satisfy a tacp object specification, whereas *t-darsl* measures satisfaction level of an entire dar w.r.t. a tacp. For this reason, we define a new measure called matrix satisfaction levels.

Definition 4.2.10 ***(msl - matrix satisfaction level):*** *Let t be a tacp and $M[r_i, o_j]$ be a cell in the roles-objects matrix corresponding to role r_j and object type o_j, the matrix satisfaction level is calculated by the msl function.*

$$msl(t, r_i, o_j) = \frac{rsl(SR_t, \{r_i\}) + (1 - d_H(ot_t, o_j))}{2} \tag{4.9}$$

Equation 4.9 calculates the matrix satisfaction level as the average of: (i) roles satisfaction level rsl (see Definition 4.2.4) between the tacp role set SR_t and a set which contains only r_i and (ii) one minus the distance $d_H()$ between the tacp object type ot_t and o_j in the object type hierarchy H.

We do not report here the code for the roles-objects matrix creation since it was already presented in [29], but we report the code necessary to find the tacp with the highest satisfaction level, i.e., code of Algorithm 4 since this is a modified version of the one presented in [29]. This new version returns the maximum satisfaction level without comparing this value with a threshold, since this value might be combined with anomaly and historical levels.

Algorithm 4: maxSatisfactionLevel()

```
    Input  : $dar$, the denied access request.
    Output: the maximum satisfaction level between the $dar$ and existing policies.
 1  T = $\emptyset$;
 2  Let $SR_d$ be the set of roles contained in dar;
 3  Let $ot_d$ be the object type contained in dar;
 4  Let $M$ be the roles-objects matrix;
 5  foreach $r_i \in SR_d$ do
 6      foreach $t_i \in M[r_i][ot_d]$ do
 7          if $t_i \notin T$ then
 8              Insert $t_i$ Into T;
 9      end
10  end
11  max = 0;
12  foreach $t_i \in T$ do
13      sl = $t\text{-}darsl(t_i, \text{dar})$;
14      if $sl > max$ then max = sl;
15  end
16  return max;
```

The 4 algorithm takes as input a denied access request *dar* and returns the maximum satisfaction level between *dar* and existing policies. First of all, the algorithm analyzes each role $r_i \in SR_d$ in the *dar* subject specification (lines 5-10). For each role r_i, the algorithm retrieves the cell in the roles-objects matrix M at row r_i and column ot_d (i.e., the object type in *dar*), which contains a subset of tacps close to be satisfied by the role r_i and by the object type ot_d. For each tacp $t_i \in M[r_i][ot_d]$ (lines 6-9), if the tacp is not already contained in T (line 7), it is inserted into the set T (line 8).

Then, for each tacp $t_i \in T$ (lines 12-15), the algorithm calculates the satisfaction level *sl* between t_i and *dar* using *t-darsl* algorithm (line 13) and if *sl* is the maximum, the algorithm stores its value in *max* variable (line 14). Once all, the tacps have been analyzed, the algorithm returns *max* (line 16).

4.3 Anomaly Based Analysis

The *anomaly based analysis* exploits anomaly detection techniques and complex event processing (CEP) in order to detect anomalous events. Once an anomaly has been detected this information is stored in the *anomaly repository*. Once an access request is denied by regular policies the system tries to find an anomaly in the *anomaly repository* which might be correlated to the denied access request; this phase is called *anomaly correlation*. The anomaly correlation is performed exploiting a pre-defined correlation between the object type of the denied access request and the event type of the anomaly. These correlations might be defined either manually by the system administrator or automatically with techniques for *correlation discovery*.

In light of these considerations, this section is structured as follows: first of all, we show how the system performs the *anomaly detection*; then, we explain how anomalies are linked to denied access requests during the *anomaly correlation* phase; finally, we explain *correlation discovery* techniques used to pre-calculate correlations used in the anomaly correlation phase.

4.3.1 Anomaly Detection

The *anomaly detection* is performed combing CEP and anomaly detection techniques. More precisely, each input or output stream (i.e., the result of processing single or multiple input streams) that we want to monitor in a CEP system is connected to a set of operators which perform the detection of anomalous events. These operators are an aggregation and a filter. The aggregate operator computes aggregations of events coming form an input or an output stream over a moving window according to an anomaly detection function *adf*(). Moreover, the operator groups output events based on a particular input field.[6]

The generic anomaly detection function *adf*() takes as input the set of events in the aggregation window and returns a score between 0 and 1 which represents the anomaly level of analyzed events, i.e., how much these events are anomalous w.r.t. the normal behavior of the system. In order to keep the presentation as general as possible, we do not define the anomaly detection function *adf*(), but in our experiments, we adopt two different techniques [56, 11] whose details are provided in Chapter 2.

Once the anomaly level has been calculated, if this level is greater than a predefined threshold, it means that analyzed events are significantly different form the normal behavior of the system, thus they should be stored as anomalies. The formal definition of anomaly is the following.

Definition 4.3.1 ***(Anomaly):*** *An anomaly is a tuple (E, et, ts, al, id), where E is the set of anomalous event(s) detected using an anomaly detection function adf(), et is the event type of the anomalous event(s), ts is the timestamp of the first anomalous event, al is the anomaly level calculated using adf() and id is the value of the identifier attribute in the anomalous event(s).*

[6] Events are always grouped based on the identifier attribute for instance the patient id, i.e., a window is created for *patient1*, a window for *patient2*, etc. In this way, all the events in the window share the same identifier value.

The event flow which realizes the anomaly detection is depicted in Figure 4.3.

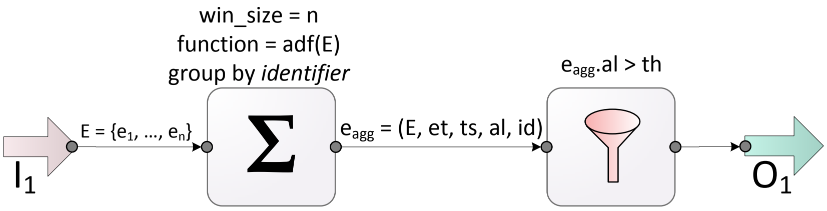


Figure 4.3: Anomaly Detection Event Flow

This detects anomalies among events received from the input stream I_1 whose event type is ET_1[7]. First of all, events received from I_1 are aggregated using a window of size n and grouped by the identifier value, i.e., the aggregation function $adf()$ is performed every n events which shares the same identifier. The aggregation operator produces an aggregated event e_{agg} which contains the anomaly level of events in E along with other information, i.e., the same information stored in an anomaly (see Definition 4.3.1). Then, the e_{agg} event is sent to a filter which discards those aggregated events whose anomaly level is lower than a predefined threshold th. The events that are not discarded are sent to output stream O_1 and stored as anomalies in the *Anomaly Repository*.

Example 4.3.1 *Consider the patient remote monitoring scenario presented in Example 4.2.1. To detect anomalies in this scenario a CEP system might be similar to the one depicted in Figure 4.3, where the input stream is connected to the event type* `VitalSigns` *presented in Example 4.2.1 and the aggregation windows size is set to 64 events. Suppose a set of 64 events E has been received from* `VitalSigns` *and the first event has been detected at 04:45AM 05-08-2012 and every event is coming from patient 1. The aggregate operator returns a high anomaly level (e.g., 0.8), since they are significantly different from the normal values (i.e., 80 bpm). Thus, the aggregated event* e_{agg} *= (E,* `VitalSigns`*, 05-08-2012 04:45AM, 0.8, 1) is sent to the filter which checks if 0.8 is greater than the predefined threshold th. If the check succeeds, the aggregated event is stored in the anomaly repository as follows:*

```
Anomaly1 {
  E = { e1,...,e64 };
```

[7] The input stream I_1 might be connected either to an input or an output stream, thus it may be the result of processing multiple input streams.

```
  et = VitalSigns;
  ts = 05-08-2012 04:45AM;
  al = 0.8;
  id = 1;
}
```

The `Anomaly1` *contains the set of anomalous events E, the event type of the anomalous events, i.e.,* `VitalSigns`*, the timestamp* "05-08-2012 04:45AM" *of the first anomalous event and the identifier value, in this case, since the identifier of event type* `VitalSigns` *is the attribute* `patient_id`*, its value is 1.*

4.3.2 Correlation Discovery

The relation between a dar and an anomaly is based on a pre-defined correlation between the object type of the dar and the event type of the anomaly (see Definition 4.3.1 for further details). The correlation is defined over an attribute in the event type and another attribute in the object type. The formal definition of the Event Type - Object Type Correlation (ETOTC) is the following.

Definition 4.3.2 ***Event Type - Object Correlation (ETOTC):*** *An etotc is represented as a tuple (et, ot, oa), where et is an event type and ot is the correlated object type. The correlation is ensured by a connection between the identifier of et and the oa attribute which belongs to the schema of ot.*

Example 4.3.2 *Consider event and object type presented in Example 4.2.1, i.e.,* `VitalSigns` *and* `MedicalRecord`*. The correlation between them might be ensured by the following ETOTC.*

```
etotc1 {
  et = VitalSigns;
  ot = MedicalRecord;
  oa = patientid;
}
```

The correlation is ensured by a connection between the identifier of VitalSigns *(i.e., the attribute* patient_id*) and the attribute* "patientid" *belonging to the schema of* MedicalRecord.

These correlations might be defined either manually by the system administrator or automatically with techniques for *correlation discovery* presented in this section. The correlation discovery problem can be defined as the process of identifying relationships between attributes of object types and attributes of event types, i.e., the event type identifiers. This is very similar to another problem known in literature as *foreign key discovery*. This is the process of discovering the set foreign keys within a database schema [71]. A foreign key represents a relationship between an attribute in a referencing table and another attribute (i.e., the primary key) in a referenced table. Indeed, the *foreign key discovery* problem and our problem are similar since they both aim finding relationship between attributes.

A large number of foreign key discovery techniques are based on the calculation of inclusion dependencies (INDs). An inclusion dependency $A \subseteq B$ means that all values of the dependent attribute A are contained in the value set of the referenced attribute B [10]. An IND between two attributes is a precondition for foreign key discovery, thus couples of attributes with INDs are candidates for foreign keys. In our model, we exploit inclusion dependency discovery algorithms for detecting INDs between attributes in event and object types. More precisely, we determine all INDs using the SPIDER (Single Pass Inclusion DEpendency Recognition) algorithm [10].

Then, among retrieved INDs, we identify whether the IND represents a ETOTC or not based on a set of features. These features are similar to the one used in foreign key discovery to detect which IDNs represent foreign keys. These features have been extensively analyzed in foreign key discovery literature using common sense and by carefully studying positive and negative examples [71]. In our model, we make use of a subset of these features in order to calculate the *correlation level*. The object attribute with the highest *correlation level* with an event attribute is stored in an ETOTC. The subset of features used in our model are called *coverage*, *colNameSim* and *multiRefRatio*. These measures are used to calculate the correlation between two attributes a_1 and a_2: *coverage* measures the ratio of values in a_1 that are contained in a_2; the *multiRefRatio* measures how often values in a_1 appears as referenced attribute in a_2; the *colNameSim* measures the similarity between the two attributes names. In the following, we provide details on how to calculate the *correlation level*.

Definition 4.3.3 ***(correlation level):*** *Let et be an event type and et.id be its identifier. Let ot be an object type and ot.a be one of its attributes. Let $V_{et.id}$ and $V_{ot.a}$ be sets of values of et.id and ot.a, respectively. The correlation level between et.id and ot.a is defined as follows:*

$$corrLev(et.id, ot.a) = f(coverage(V_{et.id}, V_{ot.a}) + colNameSim(et.id, ot.a) + multiRefRatio(V_{et.id}, V_{ot.a}) \tag{4.10}$$

The correlation level is calculated as a function $f()$ of the coverage, the column name similarity and the multi reference ratio; the $f()$ function might *be a simple average or a weighted average or a more complex combination. The coverage() function calculates ratio of values in $V_{ot.a}$ that are contained in $V_{et.id}$ as follows.*

$$coverage(V_{et.id}, V_{ot.a}) = \frac{\mid V_{et.id} \cap V_{ot.a} \mid}{\mid V_{ot.a} \mid} \tag{4.11}$$

The $multiRefRatio()$ function counts how often values in $V_{ot.a}$ appears as referenced attribute in $V_{et.id}$ set as follows.

$$multiRefRatio(V_{et.id}, V_{ot.a}) = \frac{avg(count(v_1, V_{et.id}), \ldots, count(v_n, V_{et.id}))}{\mid V_{ot.a} \mid} \tag{4.12}$$

The $colNameSim()$ function measures the similarity between the two attributes names $et.id$ and $ot.a$ using a generic string distance measure $dist()$ as follows.

$$colNameSim(et.id, ot.a) = 1 - dist(et.id, ot.a); \tag{4.13}$$

Example 4.3.3 *Consider the event type `VitalSigns` introduced in Example 4.2.1 and its events $e_1 \ldots e_6$ presented in Example 4.4.2. Consider also the object type `MedicalRecord` presented in Example 4.2.1 and its objects presented in the following.*

o_1=*{* `(patientid,1),(name,Mark),(lastname,Smith),(age,33)` *}*
o_2=*{* `(patientid,2),(name,John),(lastname,Nolan),(age,19)` *}*
o_2=*{* `(patientid,3),(name,Bob),(lastname,Palmer),(age,34)` *}*

The identifier of `VitalSigns`, i.e., `patient_id` and the `patientid` attribute of object type `MedicalRecord` have inclusion dependencies, since they have common values (i.e., 1 and 2), thus $\mid V_{patient\_id} \cap V_{patientid} \mid = 2$, whereas the total number of values of `patient_id` is three, i.e, 1, 2, 3, thus $\mid V_{ot.a} \mid = 3$, therefore coverage($V_{patient\_id}$, $V_{patientid}$) = 2/3 = 0.6.

In addition, values 1 and 2 appear three times in `VitalSigns` events, i.e., 1 appears in e_1, e_2 and e_3, whereas 2 appears in e_4, e_5 and e_6, but value 3 does not appear in `VitalSigns` events, thus $multiRefRatio$($V_{patient\_id}$, $V_{patientid}$) = avg(3, 3, 0)/2 = 2/2 = 1.

For the column name similarity, we calculate the distance between the two column names using the Hamming distance, thus $colNameSim$(`patient_id`, `patientid`) = 1 - 0.1 = 0.9.

In the end, if we use the average as function f, then the correlation level is (0.6 + 1 + 0.9)/3 = 0.83. Supposing this is the highest correlation value, the identifier of `VitalSigns`, i.e., `patient_id` and the `patient` attribute *of object type `MedicalRecord` are correlated in the ETOTC repository through the `patientid` attribute, i.e., a tuple (`VitalSigns, Medical Record, patientid`) is added to the repository.*

4.3.3 Anomaly Correlation

The *anomaly correlation* tries to find an anomaly in the *anomaly repository* which might be correlated to a dar. As we claimed in Section 4.1, this step is important because if the system finds an anomaly correlated to an access request, it is likely that the anomaly represents the triggering event of an unspecified emergency as such the dar represents the information need necessary to manage the unspecified emergency.

For instance, suppose doctor d_1 tries to read the medical record of patient p_1, but the access request is denied by regular policies; moreover an anomalous heart beat has been detected for patient p_1 within a short time before the access request. In this case, it is likely that the anomalous event represents an unspecified tachycardia emergency and the dar represents the missing policy which should authorize doctor d_1 to access patient p_1 medical record.

Once the ETOTC correlations have been defined as explained in Section 4.3.2, these are used during the *access request analysis* in order to find an anomaly which might be correlated to a dar in the *anomaly repository.* First of all, the system analyzes anomalies occurred in a specified time window before the dar. For each anomaly a_i the system checks if there is a correlation between the object type ot in the dar object specification and the event type et of the anomaly. If a correlation (et, ot, oa) has been found, the system retrieves values of identifier of the anomaly $a_i.id$ and attribute oa, if these values match it means a_i and dar are correlated. Among anomalies correlated to the dar, the system selects the anomaly with the highest anomaly level and this level is returned as the result of the *anomaly correlation* phase.

Example 4.3.4 *Consider the denied access request `dar1` presented in Example 4.2.2. Suppose, the only anomaly stored in the* anomaly repository *within specified time window before the dar is `Anomaly1` (presented in Example 4.3.1). Moreover, suppose that `etotc1` presented in Example 4.3.2 has been defined in the system. In this case, `etotc1 = (VitalSigns, MedicalRecord, patientid)` ensures a correlation between the object type `MedicalRecord` in the `dar1` object specification and the event type `VitalSigns` in `Anomaly1`. Since the value of attribute `patientid` in the dar object specification is the same of the identifier of `Anomaly1`, i.e., 1, the anomaly is related to the dar. Since this is the only anomaly correlated* *to the dar, its anomaly level, i.e., 0.8, is kept as the result of the* anomaly correlation *phase.*

The anomaly correlation presented in this section is performed by Algorithm 5 which takes as input a denied access request and the size of the window and returns the maximum anomaly level among anomalies correlated to the dar occurred in the specified window before the dar.

Algorithm 5: maxCorrelatedAnomaly()

```
   Input  : dar, the denied access request; w, the window size.
   Output: The maximum anomaly level among anomalies correlated to dar.
1  Let ETOTC be the Event Type - Object Type Correlation repository;
2  Let A be the anomalies repository; max = 0;
3  foreach $a_i \in A$ do
4      if $dar.ts - w \leq a_i.ts \leq dar.ts$ then
5          att = correlatedObjAttribute ($a_i$, dar.$o_d$.$ot_d$, ETOTC);
6          if $att \neq \emptyset$ then
7              if $a_i.id = dar.obj.att$ then
8                  if $a_i.al > max.al$ then max = $a_i$;
9              end
10         end
11     end
12 end
13 return max;
```

First of all, the algorithm analyzes each anomaly a_i in the anomaly repository A (lines 3-12). If the anomaly a_i has occurred within a time window of size w before the occurrence of *dar* (line 4), then the algorithm correlatedObjAttribute is called to find an attribute which connects the anomaly to the object of the dar in the Event Type - Object Type Correlation repository *ETOTC* (line 5).[8] If the attribute exists (line 6), then the algorithm checks if the value of the anomaly identifier *id* is equal to the attribute value in the dar object (line 7). If this is true, the algorithm checks if the anomaly level of a_i is the maximum and, in this case, it stores a_i in *max* variable. Once all, the anomalies are analyzed, the algorithm returns *max* (line 13).

4.4 Historical Based Analysis

The third strategy exploited in the *access request analysis* is the *historical based analysis*, which measures how much a denied access request is similar to one of the previously permitted dars. Every time a dar d_i is authorized

[8] The correlatedObjAttribute takes as input an anomaly a, a dar object obj and the ETOTC repository, finds in ETOTC a tuple ($a.et$, obj, oa) and returns oa if the tuple has been found, $\emptyset$, otherwise.

according to policy and/or anomaly based analysis a pair (d_i, a_i), called *controlled violation*, is stored in the *historical repository*. The pair contains the dar d_i and the anomaly a_i related to d_i, if any.[9] Controlled violations, stored in the historical repository, are used during the access request analysis.

More precisely, every time an access request is denied, the system performs the *anomaly correlation* in order to find the anomaly which is the most correlated to the dar, then it compares the dar itself and the related anomaly against controlled violations stored in the historical repository. More specifically, suppose an access request d_1 has been denied and the anomaly correlation has identified the related anomaly a_1, this pair (d_1, a_1) is compared against each controlled violation (d_i, a_i) stored in the historical repository. This comparison measures the similarity between (d_1, a_1) and (d_i, a_i) based on the similarity between d_1 and d_i, called *dars similarity level*, and the similarity between a_1 and a_i, called *anomaly similarity level.*

These two similarities are calculated separately, then results are combined together in the overall *similarity level* according to a predefined function, e.g., average. The maximum *similarity level* is returned as the result of the historical based analysis. If the current dar is not related to any anomaly or the controlled violation does not contain any anomaly, the anomaly similarity level is not calculated and the historical based analysis relies only on the dars similarity level.

It is worth noting that the historical based analysis is a refinement of policy and anomaly based analysis, which is used to enhance detection of unspecified emergencies, but it is unlikely that this analysis might identify unspecified emergencies not detected by the other strategies.

Dar Similarity Level

The *dar similarity level* measures how much two dars are similar to each other. The calculation of *dar similarity level* is similar to the satisfaction level measure presented in Section 4.2. Satisfaction level is used to compare a dar against a tacp, whereas the *dar similarity level* is used to compare two dars, but since structures of tacps and dars are similar, the two measures are similar too. More precisely, the *dar similarity level* is calculated by the *dar-sim* function whose definition is summarized in Figure 4.4.

[9] In case, an anomaly has not been found, the controlled violation contains only the dar.

dars similarity level

$$dar\ sim(d_1, d_2) = \frac{sbjsim(s_{d_1}, s_{d_2}) + objsim(o_{d_1}, o_{d_2})}{2}$$

subject and object similarity level

$$sbjsim(s_{d_1}, s_{d_2}) = \frac{w_1 * rsim(\mathrm{SR}_{\mathrm{d}_1}, SR_{d_2}) + w_2 * assim(SA_{d_1}, SA_{d_2})}{2}$$

$$objsim(o_{d_1}, o_{d_2}) = \frac{w_1 * otsim(ot_{d_1}, ot_{d_2}) + w_2 * assim(OA_{d_1}, OA_{d_2})}{2}$$

Figure 4.4: Tacp - Dar Satisfaction Level Measure

As shown in Figure 4.4 the dars similarity level $dar - sim(d_1, d_2)$ between two dars d_1 and d_2 is calculated as the average of subject similarity level $sbjsim(s_{d_1}, s_{d_2})$ and object similarity level $objsim(o_{d_1}, o_{d_2})$. Recursively, subject and object similarity level are calculated as the weighted average of role similarity level $rsim(SR_{d_1}, SR_{d_2})$ and subject attributes similarity level $assim(SA_{d_1},$
$SA_{d_2})$ and object type similarity level $otsim(ot_{d_1}, ot_{d_2})$ and object attributes similarity level $assim(OA_{d_1}, OA_{d_2})$, respectively.

Subject and object similarity levels are similar to satisfaction levels, thus we gave a brief overview of these functions. Moreover, object type similarity is calculated in the same way of object type satisfaction level, thus *otsim* function is the same of *otsl* presented in Definition 4.2.4. On the contrary, roles, subject and object attributes similarity levels are slightly different, thus we give the formal definitions of these measures.

Definition 4.4.1 ***(roles similarity level):*** *Let d_1 and d_2 be two dars, where SR_{d_1} and SR_{d_2} denote the set of roles. Let H_r, be the hierarchies for roles. The role similarity level between d_1 and d_2, denoted as $rsim(d_1, d_2)$ is defined as follows:*

$$rsiml(d_1, d_2) = 1 - min(d_{H_r}(sr^1_{d_1}, sr^1_{d_2}), \ldots, d_{H_r}(sr^n_{d_1}, sr^m_{d_2})) \forall\ sr^i_{d_1} \in SR_{d_1} \wedge sr^j_{d_2} \in SR_{d_2} \tag{4.14}$$

The role similarity level is defined as one minus the minimum distance between each role assigned to the user in d_1 and each role assigned to the user in d_2.

Since dars subject and object specification are both sets of couple attribute-value, the subject and object attributes similarity levels are calculated using the same function called *assim* (attribute set similarity level) which measure how much two sets of attributes values are similar. First, we start defining how measure the similarity of a single attribute a in dar w.r.t. to the entire set of attributes A in another dar. Then, we iterate this measure to calculate the distance between two set of attributes in two different dars.

Definition 4.4.2 ***(attribute similarity level):*** *Let d_1 and d_2 be two dars. Let a be an attribute belonging to the schema of the subject/object in d_1 with its value v and A be a set of pairs (a_i, v_i) denoting attribute names and corresponding values of the subject/object in d_2. The attribute similarity level of a on A, is calculated by the asim function.*

$$asim(a, A) = \begin{cases} 0 & if \ \nexists \ (a_i, v_i) \in A \mid a_i = a \\ 1 - d_H(v, v_i) & if \ \exists \ (a_i, v_i) \in A \mid a_i = a \\ & \wedge \ a \ is \ categorical \\ 1 - \frac{|v - v_i|}{|Dom(a)|} & otherwise \end{cases} \tag{4.15}$$

According to Equation 4.15, the attribute similarity level is equal to: (i) zero, in case the attribute a is not included in the attributes in A; (ii) one minus the d_H distance between v and v_i in a given hierarchy H, in case the categorical attribute a_i is included in the attributes A; (iii) one minus the normalized Euclidean distance between v_p and v_i in $Dom(a)$ (the domain of attribute a), otherwise.

The attribute similarity level can be easily extend to measure the similarity between two sets of couples attribute-value belonging to two different dar subject/object specifications.

Definition 4.4.3 ***(attribute set similarity level):*** *Let d_1 and d_2 be two dars. Let A_{d_1} be the set of pairs (a_i, v_i) denoting the n attribute names and corresponding values of the subject/object in d_1 and let A_{d_2} be the set of attribute names and corresponding values of the subject/object in d_2. The similarity level between A_{d_1} and A_{d_2}, is calculated by the assim function.*

$$assim(A_{d_1}, A_{d_2}) = \frac{asim(a_1, A_{d_2}) + \cdots + asim(a_n, A_{d_2})}{n} \tag{4.16}$$

As shown in Equation 4.16, the attribute set similarity level between two sets of attributes A_{d_1} and A_{d_2} is calculated as the average of the attribute similarity levels between each attribute $a_i \in A_{d_1}$ and A_{d_2}.

The *assim* is used in computation of both subject and object attributes similarity levels, as shown in Figure 4.4.

Example 4.4.1 *Consider the denied access request* `dar1` *presented in Example 4.2.2 and the following* `dar2` *whose access request has been performed by* `paramedic2` *belonging to the geriatric ward which tried to access medical record of* `patient2`*.*

```
dar2 {
  $s_{id}$ = paramedic2
  $o_{id}$ = MedialRecord2
  $s_{dar_2}$ = {
    $SR_{dar_2}$ = {paramedic}
    $SA_{dar_2}$ = {(ward, Geriatric Ward)}
  }
  $o_{dar_2}$ = {
    $ot_{dar_2}$ = MedicalRecord
    $OA_{dar_2}$ = {(patientid, 2)}
}
}
```

As explained in Example 4.2.2, we suppose roles, object types and attributes are hierarchically organized as shown in Figure 4.2. In this case, the role and object similarity levels are calculated as follows: (1) $rsim(dar_1, dar_2)$ *= 1, since the two paramedics have the same role; (2)* $otsim(dar_1, dar_2) = 1$*, since the two objects have the same type.*

The set of subject attributes in dar_1 *is* $SA_{dar_1} = \{$(ward, Cardiac Ward)$\}$*, whereas the set of subject attributes in* dar_2 *is* $SA_{dar_2} = \{$(ward, Geriatric Ward)$\}$*. Since there is just one attribute in the subject specification the subject attribute similarity level correspond to the attribute similarity level* $asim(ward, SA_{dar_2})$ *between* `ward` *and* SA_{dar_2}*. Since* `ward` *is a hierarchically organized attribute, the similarity is calculated as* $1 - d_H($Geriatric Ward, Cardiac Ward$)$*= 1 - 0.33 = 0.67.*

The set of object attributes in dar_1 *is* $OA_{dar_1} = \{(patientid, 1)\}$*, whereas the set of object attributes in* dar_2 *is* $SA_{dar_2} = \{(patientid, 2)\}$*. Since there is only one attribute in the object specification, the object attribute similarity level correspond to the attribute similarity level* $asim(patientid, OA_{dar_2})$ *between* `patientid` *and* OA_{dar_2}*. Since* patientid *is a numerical attribute, the similarity is calculated as* $1 - \frac{|2-1|}{10}$ *= 1 - 0.1 = 0.9.*[10]

Combining these values, it is possible to calculate subject and object similarity levels as follows:

[10] In this example, we suppose the total number of patient is 10.

(1) $sbsim(s_{dar_1}, s_{dar1}) = \frac{1+0.67}{2} = 0.835$.
(2) $objsim(o_{dar_1}, o_{dar1}) = \frac{1+0.9}{2} = 0.95$. *The overall dars similarity level is calculated as the average of these two values:* $dar-sim(dar_1, dar_2) = \frac{0.835+0.95}{2} = 0.8925$.

Anomaly Similarity Level

The *anomaly similarity level* measures how much events causing an anomaly are similar to events causing another one. The similarity between two sets of events is calculated as the average of similarity between each event in one set and each event in the other one. In order to calculate the similarity level between two anomalies a_1 and a_2, we need to measure how much events causing a_1 are similar to events causing a_2. Obviously, events must belong to the same event type. More precisely, given two events sets: E_1 and E_2 we need to measure how much each event $e_i \in E_1$ are similar to events in E_2. Then, we iterate this measure to calculate the similarity between the two set of events E_1 and E_2. We start defining similarity between two events then between an event and an event set and finally between two event sets.

Definition 4.4.4 ***(single event similarity level):*** *Let e_1 and e_2 be two events. Let A_{e_1} be the set of pairs $(a^i_{e_1}, v^i_{e_1})$ denoting attribute names and corresponding values of e_1 and let A_{e_2} be the set of pairs $(a^i_{e_2}, v^i_{e_2})$ denoting attribute names and corresponding values of e_2. The single event similarity level is calculated as follows.*

$$sesl(e_1, e_2) = assim(A_{e_1}, A_{e_2}) \tag{4.17}$$

Since events are similar to user/object types because they are both represented as a set of couples attribute-value, we exploit attribute set similarity (Definition 4.4.3) to calculate the single event similarity.

The single event similarity definition can be easily extended to measure the similarity between an event and a set of events.

Definition 4.4.5 ***(multiple event similarity level):*** *Let e be an event. Let E be a set of events $\{e_1, \ldots, e_n\}$. The similarity level between e and E, is calculated by the mesl function.*

$$mesl(e, E) = \frac{sesl(e, e_1) + \cdots + sesl(e, e_n)}{n} \tag{4.18}$$

The similarity level between e and E, is calculated as the average of $sesl(e, e_j)$ for each $e_j \in E$.

The multiple event similarity definition can be easily extend to measure the similarity between two event sets.

Definition 4.4.6 ***(event set similarity level):*** *Let E_1 be a set of events $\{e_1, \ldots, e_n\}$ and E_2 be a another set of events. The similarity level between E_1 and E_2, is calculated by the essl function.*

$$essl(E_1, E_2) = \frac{mesl(e_1, E_2) + \cdots + mesl(e_n, E_2)}{n} \tag{4.19}$$

The similarity level between E_1 and E_2, is calculated as the average of multiple similarity levels calculated between each event $e_j \in E_1$ and E_2.

It is important to note that the identifier attribute is not considered in Equation 4.19, because it is useless in the event set similarity. This is clarified in the following example.

Example 4.4.2 *Consider the `Anomaly1` presented in Example 4.3.1 and suppose it contains the following set of events.*

`E_1 {`
e_1`={(heart_rate,102),(glucose_level,120),(patient_id,1)}`
e_2`={(heart_rate,103),(glucose_level,120),(patient_id,1)}`
e_3`={(heart_rate,101),(glucose_level,120),(patient_id,1)}`
`}`

Moreover, suppose that the current access request analysis has found another anomaly in the historical repository *which contains the following set of events.*

`E_2 {`
e_4`={(heart_rate,104),(glucose_level,120),(patient_id,9)}`
e_5`={(heart_rate,102),(glucose_level,120),(patient_id,9)}`
e_6`={(heart_rate,100),(glucose_level,120),(patient_id,9)}`
`}`

These two sets of events both represent a tachycardia emergency and their similarity is calculated as follows. Let us start with the single event similarity level between e_1 and e_4, which, according to equation 4.17, is calculated as $sesl(e_1, e_4) = assim(A_{e_1}, A_{e_4})$. Since assim is calculated as the average of the attribute similarity levels between each attribute in the two attribute sets, we begin from heart rate attribute. Since heart rate is a numerical attribute, the similarity $asim(heart\_rate, A_{e_4})$ is calculated as $1 - \frac{|102-104|}{200} = 1 - 0.01 = 0.99$[11]. Regarding the other attributes, `glucose_level` has the

[11] In this example, we suppose heart rate values are ranged between 0 and 200.

same values for e_1 *and* e_4*, thus its similarity is 1, whereas* `patient_id` *is not considered since it is the identifier of* `VitalSigns` *event type. The identifier is not considered because the distance between two identifiers is not significant, i.e., the two set of events are similar because they both represent a tachycardia emergency regardless of the identifier value. It is worth noting that considering the identifier might affect the result of this measure because the two values are significantly different.*

In light of these results, the single event similarity level $sesl(e_1, e_4) = \frac{0.99+1}{2} = 0.995$*. The single event similarity is then calculated between* e_1 *and the events in* E_2*, i.e,* $sesl(e_1, e_5) = 1$*,* $sesl(e_1, e_6) = 0.995$*. These results are combined to calculate* $mesl(e_1, E_2) = \frac{sesl(e_1,e_4)+sesl(e_1,e_5)+sesl(e_1,e_6)}{3} = \frac{0.995+1+0.995}{3} = 0.996$*.*

The multiple event similarity level is then calculated between e_2 *and the events in* E_2*, i.e.,* $mesl(e_2, E_2) = 0.993$ *and between* e_3 *and the events in* E_2*, i.e.,* $mesl(e_3, E_2) = 0.997$*. Finally, these results are combined to obtain the overall event set similarity* $essl(E_1, E_2) = \frac{mesl(e_1,E_2)+mesl(e_2,E_2)+mesl(e_3,E_2)}{3} = \frac{0.996+0.993+0.997}{3} = 0.9953$*.*

Finally, we can define *historical level* by combing the above measures.

Definition 4.4.7 ***(Historical level):*** *Let* (d_1, a_1) *be a controlled violation. Let* d_2 *be a denied access request and* a_2 *be the most related anomaly. The historical level is calculated by the* hl *function.*

$$hl((d_1, a_1), (d_2, a_2)) = \frac{w_1 \times dar - sim(d_1, d_2) + w_2 \times essl(E_{a_1}, E_{a_2})}{2} \quad (4.20)$$

Equation 4.20 calculates historical level as the weighted average between the dars similarity level $dar - sim$ *(Figure 4.4) and the event set similarity level* $essl$ *(Definition 4.4.6).*

Example 4.4.3 *Consider the dars similarity level calculated in Example 4.4.1, i.e.,* $dar - sim(dar_1, dar_2) = 0.8925$ *and the event set similarity level computed in Example 4.4.2, i.e.,* $essl(E_1, E_2) = 0.9953$*. The overall historical level might be calculated as the average of these two values, i.e.,* $\frac{0.8925+0.9953}{2} = 0.9439$*. This level is returned as the result of the* historical based analysis *phase.*

The historical based analysis presented in this section is performed by Algorithm 6 which takes as input a denied access request and the related anomaly and returns the maximum similarity level between the couple (dar, a) and previously permitted access requests and related anomalies.

Algorithm 6: maxControlledViolationsHistoryLevel()

```
  Input  : $dar$, the denied access request, $a$, the related anomaly
  Output: The maximum similarity level between $dar$ and previously permitted
          access requests.
1 Let $H$ be the historical repository;
2 max = 0;
3 foreach $(dar_i, a_i) \in H$ do
4     sim = hl (($dar_i$, $a_i$), (dar, a));
5     if $sim > max$ then max = sim;
6 end
7 return max;
```

First of all, the algorithm analyzes each couple (dar_i, a_i) of previously permitted access requests and related anomalies in the historical repository H (lines 3-6). If the historical level hl between (dar_i, a_i) and (dar, a) calculated using Equation 4.20 (line 4) is the maximum (line 5), then the algorithm stores its value in max variable (line 5). Once entries in the historical repository have been analyzed, the algorithm returns the maximum historical level max (line 13).

Chapter 5

Enforcement

In this chapter, we show how the proposed emergency policies model can be enforced on top of a CEP system. More precisely, we implemented a prototype framework in Java on top of a StreamBase CEP platform [75]. The framework is called SHARE (Secure information sHaring frAmework for emeRgency managemEnt). The architecture which support the core model and its extensions is presented in Section 5.1, while the extend framework for the support of unspecified emergencies is presented in Section 5.2.

5.1 Architecture

The prototype architecture is shown in Figure 5.1. The main module of this architecture is the *Emergency Handler* which performs registration and enforcement of emergencies and polices. Given an emergency policy, it is necessary to register *init* and *end* events in the CEP, as well as emergencies and polices information in the repositories. The *Emergency Repository* contains emergency descriptions, whereas the *Tacp templates repository & acp repository* contains tacp templates and regular access control polices. An emergency definition in the emergency repository is stored as a tuple $(init, end, timeout, identifier, obl)$ as explained in Definition 3.1.14 and a tacp template in the tacp templates repository & acp repository is stored as a tuple $(sbj, obj, priv, exp, obl)$ as explained in Definition 3.2.1. When a user makes an access request, its profile is loaded from the *User Profiles Repository.* Each user profile contains authentication information such as user name and password and user attributes such as user role and personal information, e.g., name, family name, age etc.

In order to explain the prototype details, we analyze how it works during the three most important phases: (1) specification of emergency polices and

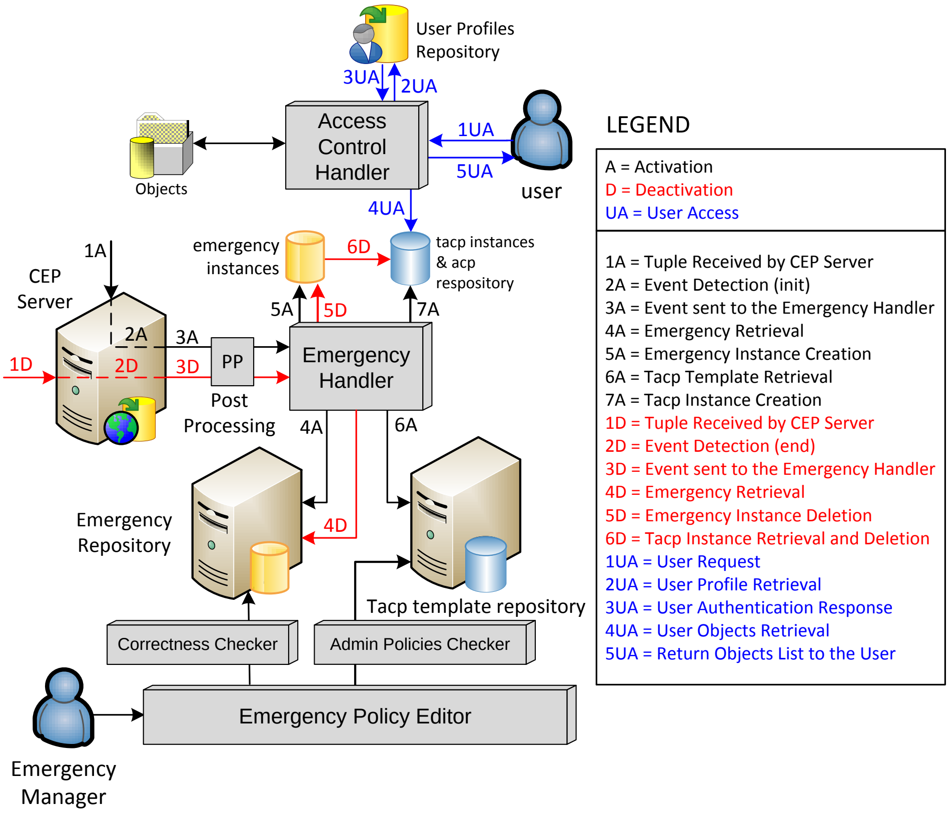

User Profiles Repository
3UA
2UA
Access Control Handler
Objects
1UA
5UA
user
4UA
LEGEND
A = Activation
D = Deactivation
UA = User Access
1A = Tuple Received by CEP Server
2A = Event Detection (init)
3A = Event sent to the Emergency Handler
4A = Emergency Retrieval
5A = Emergency Instance Creation
6A = Tacp Template Retrieval
7A = Tacp Instance Creation
1D = Tuple Received by CEP Server
2D = Event Detection (end)
3D = Event sent to the Emergency Handler
4D = Emergency Retrieval
5D = Emergency Instance Deletion
6D = Tacp Instance Retrieval and Deletion
1UA = User Request
2UA = User Profile Retrieval
3UA = User Authentication Response
4UA = User Objects Retrieval
5UA = Return Objects List to the User
1A
CEP Server
emergency instances
6D
tacp instances & acp respository
5A
5D
7A
2A
3A
1D
2D
3D
PP
Post Processing
Emergency Handler
4A
6A
4D
Emergency Repository
Tacp template repository
Correctness Checker
Admin Policies Checker
Emergency Policy Editor
Emergency Manager

Figure 5.1: System Architecture

emergency description, (2) emergency activation/deactivation (3) user access.

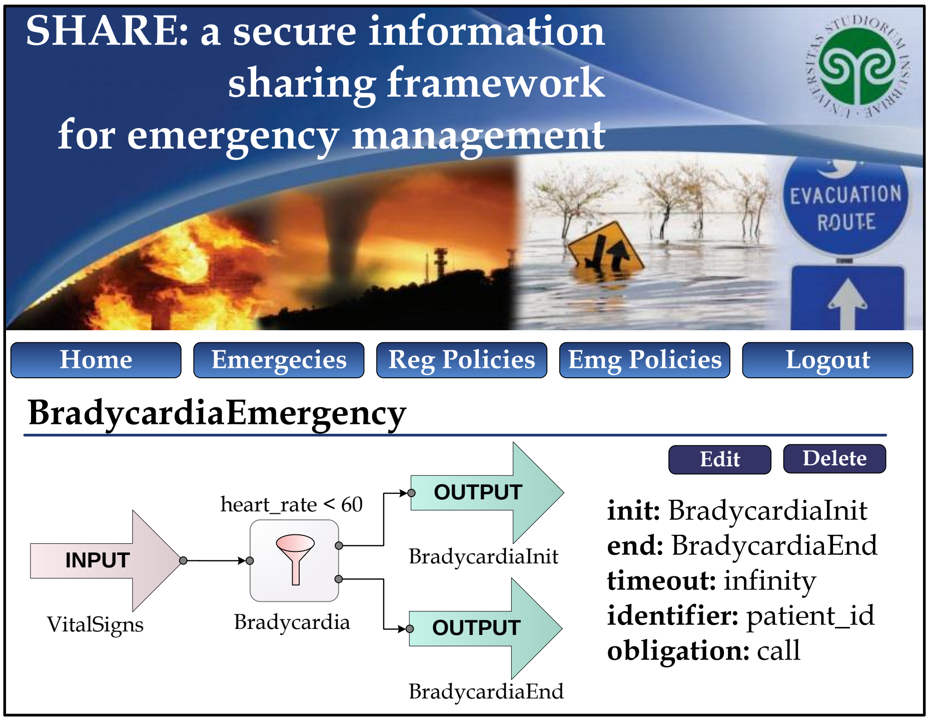
Emergency and Emergency Policy Specification: the *Emergency Policy Editor* allows the creation of emergency descriptions, tacp templates, and emergency policies. Emergency descriptions are stored into the *Emergency repository*. Before emergency registrations, correctness validity checks described in Section 3.2.1 are performed by the *Correctness Checker*. Similarly, new tacp templates are stored into the *Tacp template repository*. When an emergency manager, using the *Emergency Policy Editor*, creates/updates an emergency policy, then the administration policy enforcement is performed by the *Admin Policies Checker*. If this is successfully executed, the new/updated emergency policy is stored in the system.

Emergency Activation/Deactivation: Once the CEP server receives a tuple (1A) triggering an *init* event (2A), this is immediately sent to the Emergency Handler (3A). Before arriving to the Emergency Handler, the *Post Processing (PP)* module checks through the post-processing validity check if the tuple might cause an SHP and executes one of the response actions described in Section 3.2.1. If this is not the case, the Emergency Handler retrieves from the Emergency Repository the emergency related to the received tuple (4A), if any. Then a new emergency instance is created (5A) unless another emergency instance with the same identifier has been already created (i.e., the emergency policy is already active). Moreover, the Emergency Handler retrieves from the Tacp template repository, templates related to the activated emergency (6A), if any, by also creating the corresponding tacp instance (7A). When the CEP server receives a tuple (1D) that causes the detection of an *end* event (2D), it sends such a tuple to the Emergency Handler (3D), which checks if there exists an emergency related to it in the Emergency Repository (4D). If this is the case, the corresponding emergency and tacp instances are deleted (5D-6D).

User Access: When a user u successfully logs into the system (1UA), the Access Control Handler retrieves its profile from the User Profiles Repository (2-3UA). This contains profile attributes and the set of roles u is authorized to play. To compute the set of objects u is authorized to require, the Access Control Handler verifies each regular access control policy in place by returning objects identified by those policies whose authorized roles (i.e., roles specified in their subject specification) include at least a role assigned to u. To this set, the Access Control Handler also adds objects authorized by some temporary access control policy instances (4UA). The object contained in a tacp instance is returned if subject, object and context conditions in the tacp are satisfied. More details about conditions evaluation are provided in Chapter 6.

In the following example, we show how the emergency policy enforcement works in the patient remote monitoring scenario presented in Example 3.1.1.

Example 5.1.1 ***Emergency and Emergency Policy Specification:*** *consider an emergency manager who wants to define the* `BradycardiaEmergency` *presented in Example 3.1.14. Such emergency might be defined as depicted in the screenshot in Figure 5.2.*



SHARE: a secure information sharing framework for emergency management

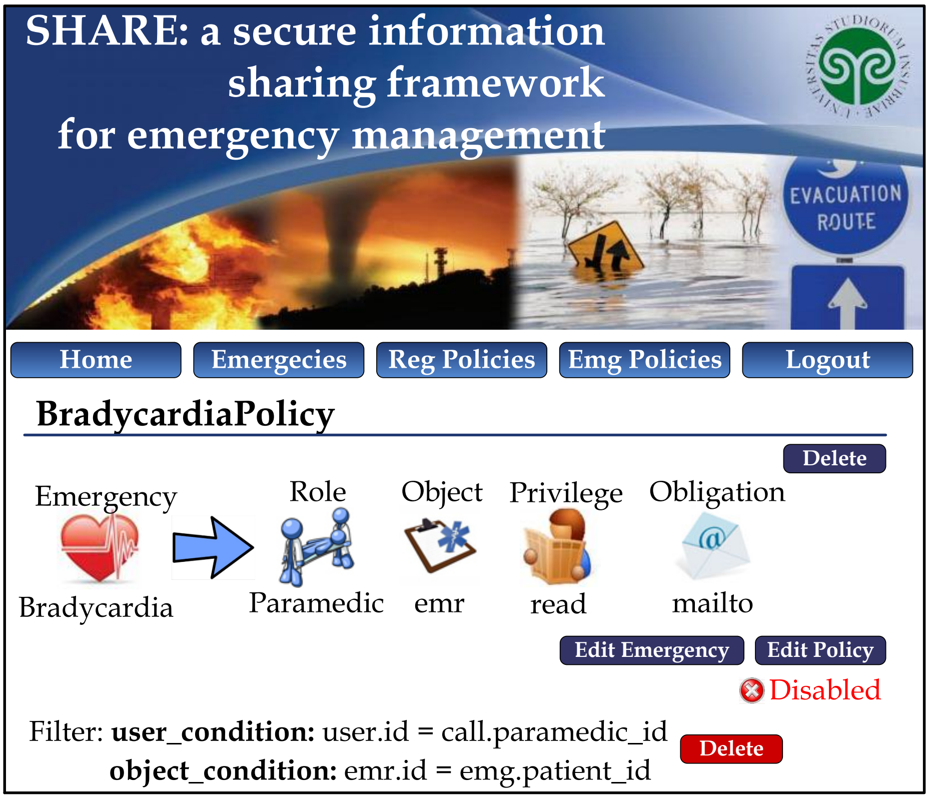
BradycardiaEmergency

init: BradycardiaInit
end: BradycardiaEnd
timeout: infinity
identifier: patient_id
obligation: call

Figure 5.2: Emergency Editor Screenshot

This screenshot shows the `BradycardiaEmergency` *where* `BradycardiaInit` *is the event which starts the emergency when the heart rate of a patient is lower than or equal to 60 bpm,* `BradycardiaEnd` *is the event which ends the emergency when the heart rate returns higher than 60 bpm,* `timeout` *is set to infinity and* `identifier` *is the attribute* `patient_id`*. The identifier patient_id guarantees the correlation between init/end events and also that different emergencies are raised for different patients.*

An emergency policy which connects Bradycardia *with the corresponding* tacp *and* obligation *might be the* BradycardiaPolicy *depicted in the screenshot in Figure 5.3.*



Filter: **user_condition:** user.id = call.paramedic_id
object_condition: emr.id = emg.patient_id

Figure 5.3: Emergency Policy Editor Screenshot

In this case the tacp extends access to the Electronic Medical Record (EMR) of the patient under emergency (Filter: $obj.id = emg.patient\_id$) to the paramedic who answered to the emergency call (Filter: $user.id = call.paramedic\_id$). The obligation mailto *ensures that when a paramedic reads the patient EMR, then an email is sent to the patient email address.*

Emergency Activation: *when the CEP server detects (2A) an event from the output stream* `BradycardiaInit`*, then it sends this event (3A) to the Emergency Handler. The Emergency Handler retrieves (4A-5A)* `BradycardiaEmergency` *and the policy template* `BradycardiaPolicy`*. Then the Emergency Handler stores the following emergency instance (6A) and tacp instance (7A).*

id	emg_id	identifier	obl
31	BradycardiaEmergency	1	call_ambulance(40 Storrow Dr)

Table 5.1: *Emergency* Instance from Example 3.1.14

id	tacp_id	sbj	obj
41	BradycardiaPolicy	role = paramedic ∧ user.id = 12	EMR.id = 1
priv	exp	obl	emg_instance_id
read	-	mailto(a@domain.com)	31

Table 5.2: *Tacp* Instance from Example 3.2.1

When the emergency is detected, a new emergency instance (id $= 31$*) is created with* $identifier = 1$ *(the patient_id of the patient under emergency) and obl* $= call\_ambulace(40$ *Storrow Dr), assuming that 40 Storrow Dr is the patient address. Then the emergency obligation is fulfilled and a tacp instance is created with subject role* $= paramedic \wedge user.id = 12$ *assuming that 12 is the identifier of the paramedic on the ambulance,* $EMR.id = 1$ *and obl* $= mailto(a@domain.com)$*, assuming that a@domain.com is the patient mail address.*

User Access: *When a user u logs into the system with the role paramedic and with id = 12 (1UA) its profile is retrieved by the Access Control Handler in the User Profiles Repository (2UA). In this case the profile contains the list* $u_{roles} = \{$ *paramedic* $\}$*. The user u is then authenticated (3UA) and the Access Control Handler analyzes the polices in Tacp templates & acp repository (4UA). In this case, the only tacp instance related with a user with the role paramedic is the one in Table 5.2, therefore the system checks the subject condition (paramedic.id = 12) and returns the objects that satisfy the object condition (5UA), i.e., the list { EMR.id = 1 } is returned to the logged user (5UA).*

Emergency Deactivation: *When the CEP server detects (2D) an event* e_2 *from the output stream* O_2*, then it sends this event (3D) to the Emergency Handler. The Emergency Handler retrieves (4D) the emergency* 12 *related to the event received and the corresponding emergency instance and tacp instance and deletes (5D-6D) them (i.e., the emergency instance* 31 *and the tacp instance* 41*).*

5.2 Unspecified Emergencies Architecture

The core model architecture has been further extended to support unspecified emergencies detection and management. The extended architecture is presented in Figure 5.4, where new modules are highlighted in a different color.

The functions performed by the new modules is explained in the following.

Anomaly Detector: this module is in charge of detecting anomalous events in data streams connected to the CEP and store these anomalies into the anomaly repository.

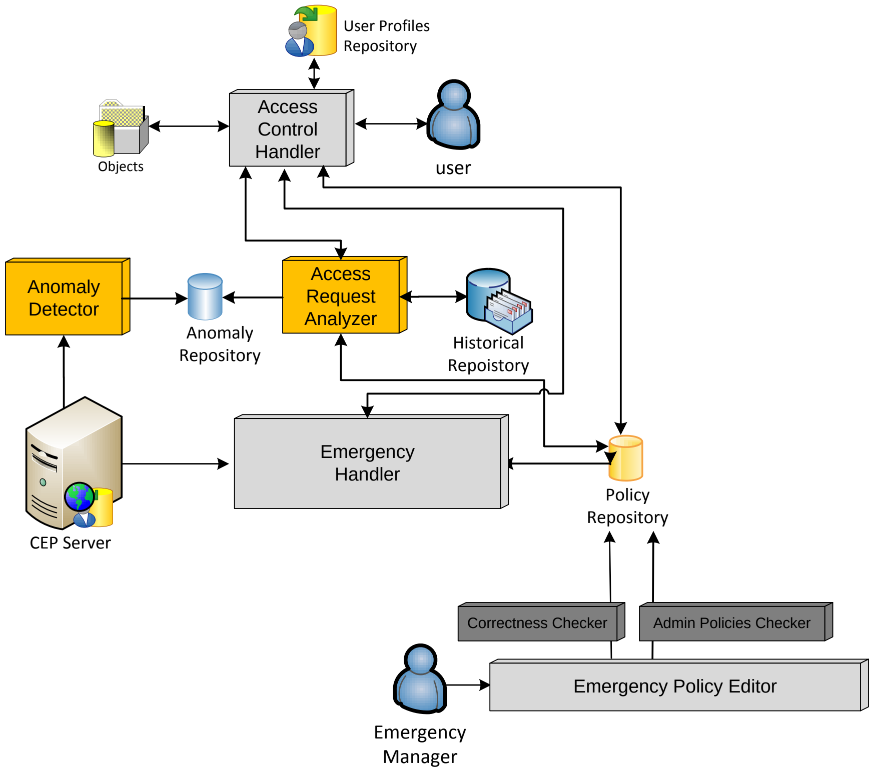


Figure 5.4: Unspecified Emergencies Framework Architecture

Access Request Analyzer: once an access request has been denied by the access control handler, the access requests analyzer performs policy, anomaly and historical based analysis. The policy based analysis is performed checking in the policy repository whether the current access request is close to satisfy existing polices. The anomaly based analysis is executed checking if the current access request is related to any of the anomalies stored in the anomaly repository. The historical based analysis is carried out searching previously permitted access requests similar to the current one. Once the satisfaction, anomaly and historical level of the current access request has been calculated, the access request analyzer decides whether authorizing or not it based on the calculated levels, threshold and tolerance values.

Chapter 6

Experiments

In this section experiments on the prototype framework present in Chapter 5 are presented. More precisely, experiments results on the framework implementing the core model and its extensions are introduced in Section 6.1, while experimental results for the detection and management of unspecified emergencies are described in Section 6.2.

6.1 Emergency Policy Evaluation

In this section, the performance results of the prototype system are discussed. The experiments were run on an Intel Core i7 2.00 GHz CPU machine with 4Gb RAM, running Windows 7. The prototype implements the architecture explained in Chapter 5, therefore we carried out tests on every step of the emergency life cycle. In this section, we report results on overall time for emergency activation/deactivation and user access time. Before presenting the experimental results, we provide details on the dataset.

6.1.1 Dataset

In order to carry out the experiments on emergency detection, activation and deactivation, we developed an *emergency events generator*. By means of this generator, we can create a specific number of *init* and *end* events by varying their complexity, which is measured in terms of number of operators (i.e., selection, aggregation and join operators) contained into the event.

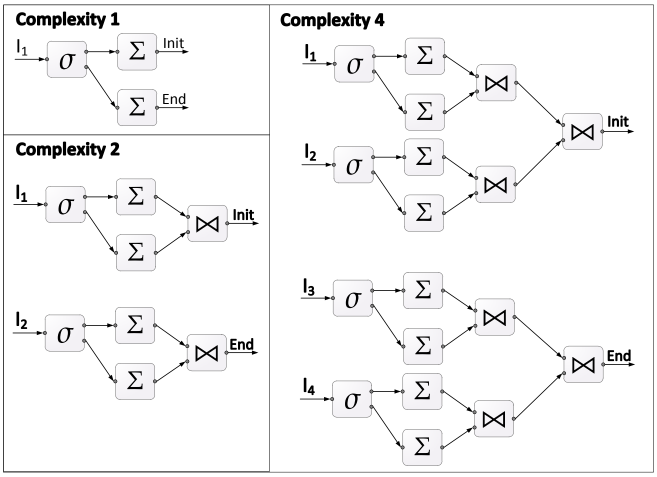


Figure 6.1: Emergency Event Complexity

As shown in Figure 6.1, in case of complexity 1, the generated event takes as input a unique stream, over which it evaluates one selection and two aggregations. From this unique input stream, it generates both *init* and *end* events. With a complexity of two, the event contains 2 input streams, 2 selections, 4 aggregations and 2 join operators. In general, in case of complexity n, the number of input streams is n, the number of selections is n, the number of aggregations is $2n$ and the number of join operators is $\sum_{i=1}^{\sqrt{n}} 2^i$ (see, as an example the case of complexity 4 in Figure 6.1). The *emergency events generator* is also able to send a certain number of tuples to input streams at a certain speed (tuples per second) so as to trigger, with a given frequency, the *init* and *end* events previously created.

Another important aspect of the dataset is the number of emergencies and tacps which is fixed and set to 100. These 100 emergencies and tacps are activated and deactivated 100 times during the experiments for a total of 10.000 emergency activations and deactivations. During experiments the tuples rate varies from 1.000 to 10.000 tuples per second, which means the number of activated emergencies per hour varies from 3.600.000 to 36 million. Considering for instance that the daily volume of 911 calls for New York city is 30.000 [19] we believe that our experimental numbers are large enough to guarantee high performance in a real emergency management system.

In the following the results of the experiments are shown for each previously defined steps: (1) event detection time, (2) emergency activation/deactivation time and (3) user access time. The first experiment is focused on demonstrating that the CEP system is a valid base for an event-based emergency management, whereas the other experiments are focused on testing the efficiency of the event handler. In the end, the performance of post-processing module are also tested.

6.1.2 Event Detection Time

The *event detection time* depends on the complexity of the emergency events registered into StreamBase and also on the *tuples rate*, i.e., the numbers of tuples received by StreamBase, measured in tuples per second.

Event Detection Time based on Event Complexity

In this experiment, the emergency events generator sends a fixed number of tuples (i.e., 100.000) to the CEP at a fixed time interval (i.e., 100 tuples per seconds) and the event detection time is measured varying the complexity from 2 to 64. This means that the *init* event varies from having two input streams, two selections, four aggregations and two join operators to 64 input streams, 64 selections, 128 aggregations and 510 join operators.

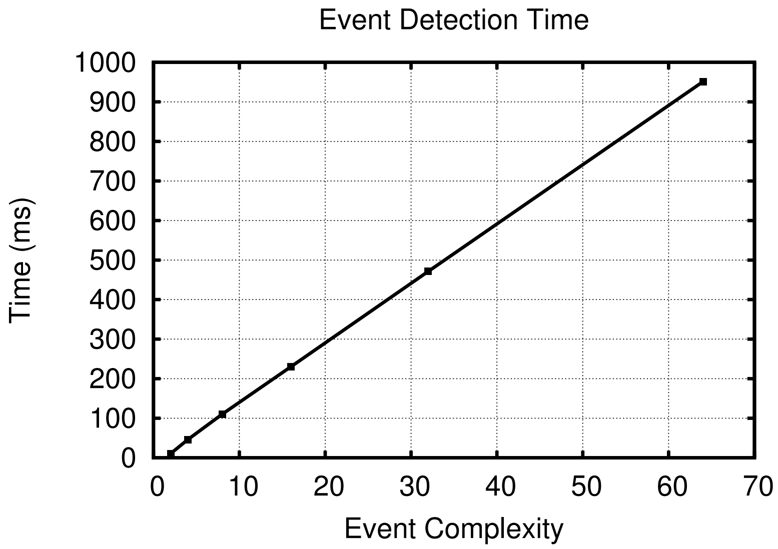


Figure 6.2: Event Detection Time

Figure 6.2 reports the event detection time: in the first case, i.e., complexity 2, the event detection time is 10 milliseconds, while in the last case, i.e., complexity 64, the event detection time is 950 milliseconds. A detection time of one second might be considered too high in critical scenario. However, it is important to note that we do not expect that emergency events have such a high complexity.

Event Detection Time based on Tuples Rate

In the following test, the emergency events generator is set up with a fixed number of emergencies (i.e., 10) with a fixed complexity (i.e., 4) and it sends a fixed number of tuples (i.e., 500, for each emergency input) to StreamBase. The event detection time is measured varying the tuples per second rate from 10 t/s to 1.000 t/s.

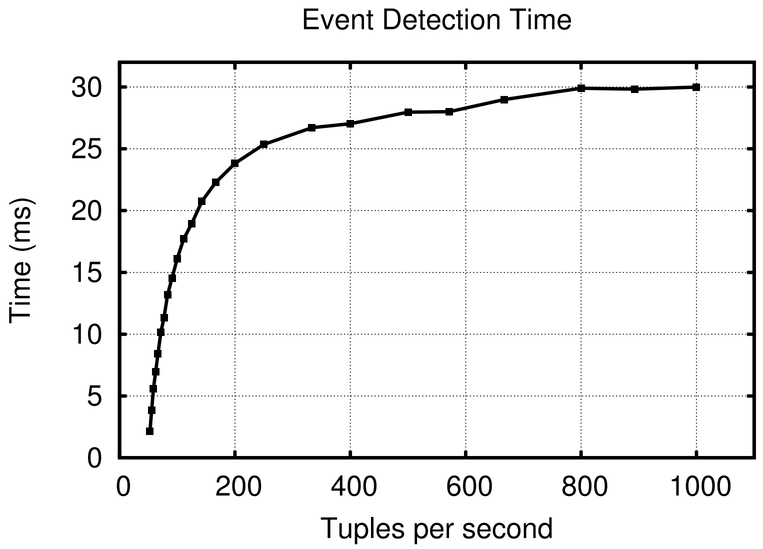


Figure 6.3: Event Detection Time (Rate)

As shown in Figure 6.3, in the first case, i.e., 10 t/s, the event detection time is 2 millisecond, while in the last case, i.e., 1.000 t/s, the event detection time is 30 milliseconds. These results show that StreamBase is scalable in the number of received tuples, therefore it is suitable for the purposes of our prototype.

Emergency Activation Time

The *emergency activation time* represents the time elapsed between the detection of the emergency by StreamBase and the effective activation of the corresponding emergency policy, that is, the time of creation of instances of the corresponding emergency and tacp.

Emergency Creation Time

The *emergency creation time* is composed of the time necessary to retrieve the emergency related to the *init* event received (4A. *emergency retrieval time*) and the time to create the corresponding emergency instance (6A. *emergency instance creation time*). In this experiment, the emergency generator sends a fixed number of tuples (i.e., 10.000) to the CEP input streams in order to trigger the related events. The emergency creation time is measured varying the tuples rate, from 1000 to 10.000, and consequently the events rate released by the CEP. The emergency instance creation time is constant (around 5 ms), whereas the emergency retrieval time grows in a linear way in the number of events per second (e/s), as shown in Figure 6.4.

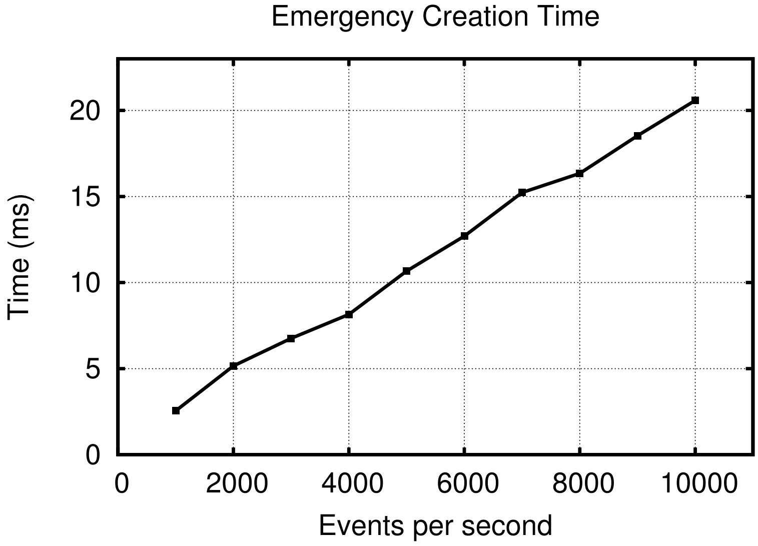


Figure 6.4: Emergency Creation Time

The emergency creation time shown in Figure 6.4 comprise both emergency instance creation time and emergency retrieval time, but since the emergency instance creation time is constant the growth depends only on the emergency retrieval time. Referring to Figure 6.4: in the first case, i.e. 1000 e/s, the emergency creation time is 2 milliseconds, while in the last case, i.e. 10.000 e/s, the emergency creation time is 21 milliseconds. These results show that the system is scalable in the number of the received events and the CPU time is efficient also with a high events rate.

Tacp Creation Time

The *tacp creation time* is composed of the time necessary to retrieve the tacp template related to the emergency (5A. *tacp template retrieval time*) and the time to create the corresponding tacp instance (7A. *tacp instance creation time*). Since a tacp template is directly connected to an emergency through a many-to-many relationship, the tacp template retrieval time is constant (around 5 ms). Conversely, the tacp instance creation time depends on the number of subject and object conditions. As shown in Figure 6.5, we tested this time varying the number of subject and object conditions from 2 to 2.048.

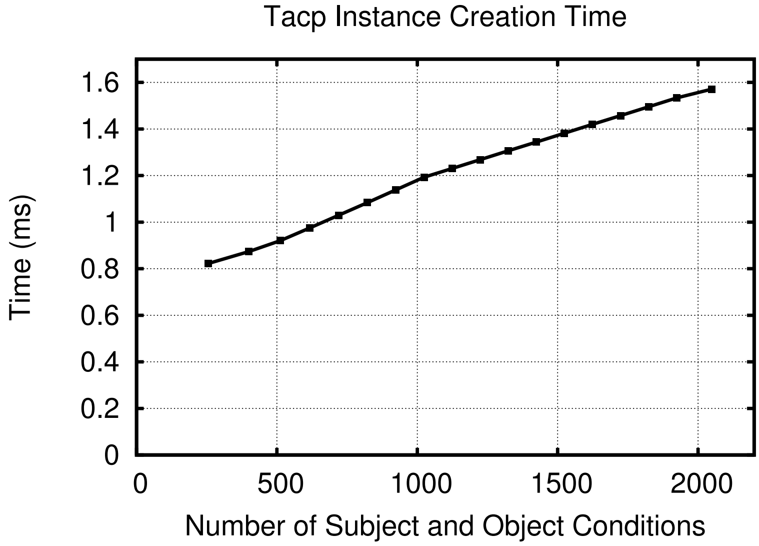


Figure 6.5: Tacp Instance Creation Time

In this case, the tacp instance creation time growth is linear in the number of subject and object conditions. In the first case, i.e. 2 subject and object conditions, the tacp instance creation time is 0,8 milliseconds, while in the last case, i.e. 2.048 conditions, the tacp instance creation time is 2,3 milliseconds. These results show that the number of conditions does not affect the system performance.

Emergency Deactivation Time

The *emergency deactivation time* represents the time elapsed between the detection (2D) of the end of an emergency by StreamBase and the effective deactivation of the corresponding emergency policy.

Emergency Deletion Time

The *emergency deletion time* is composed of the time necessary to retrieve the emergency related to the *end* event received (4D. *emergency retrieval time*) and the time to delete the corresponding emergency instance (5D. *emergency instance deletion time*). In this experiment, the *emergency generator* sends a fixed number of tuples (i.e., 10.000) to the CEP in order to trigger the related *end* events. The emergency deletion time is measured varying the events rate from 1000 to 10.000 events per second. The emergency instance deletion time is constant (around 1.7 ms), whereas the emergency retrieval time growth is linear in the number of events per second, since the emergency instance deletion time is constant the growth depends only on the emergency retrieval time.

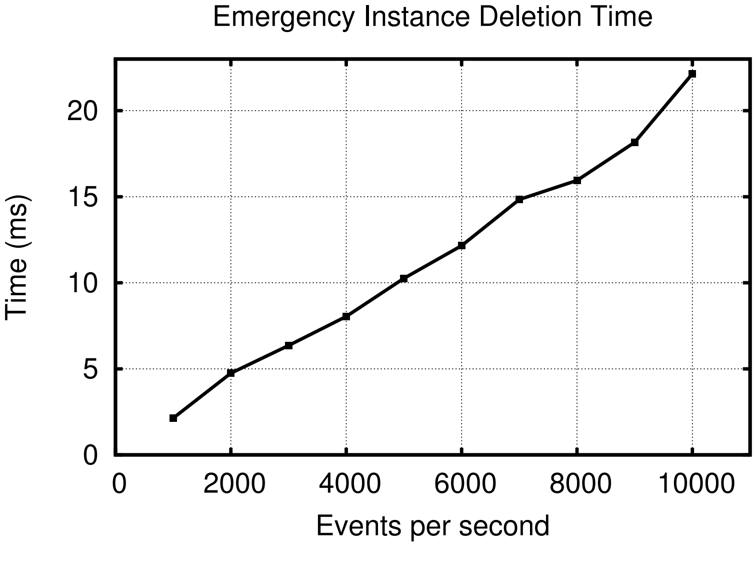


Figure 6.6: Emergency Deletion Time

Referring to Figure 6.6, in the first case, i.e. 1000 e/s, the emergency deletion time is 3 milliseconds, while in the last case, i.e. 10.000 t/s, the emergency deletion time is 20 milliseconds.

Tacp Deletion Time

The *tacp deletion time* is composed of the time necessary to retrieve the tacp instance related to the emergency instance (*tacp instance retrieval time*) and the time to delete the corresponding tacp instance (6D. *tacp instance deletion time*). Since a tacp instance is directly connected to an emergency instance through a many-to-many relationship, the tacp instance retrieval time is constant (around 1,2 ms). The tacp instance deletion time is constant too (around 1,6 ms).

Overall Activation and Deactivation Time

The *overall activation time* represents the time elapsed between the detection of an emergency and the effective activation of the corresponding emergency policy. This is given by: (i) the time needed to retrieve the emergency related to the triggered *init* event (*emergency retrieval time*), (ii) the time for the creation of the corresponding emergency instance (*emergency instance creation time*), (iii) the time necessary to retrieve the tacp template related to the emergency (*tacp template retrieval time*) and (iv) the time to create the corresponding tacp instance (*tacp instance creation time*).

The *overall deactivation time* represents the time elapsed between the detection of a tuple satisfying an *end* event and the effective deactivation of the corresponding emergency policy. Similar to activation, this is given by the *emergency retrieval time*, the *emergency instance deletion time* and the *tacp instance retrieval and deletion time.*

In this experiment, the number of emergency descriptions and tacps is fixed and set to 100. Each emergency is connected to one tacp. The emergency generator sends a fixed number of tuples (i.e., 200) to StreamBase input streams: the first 100 tuples activate the 100 emergencies and related 100 tacps, whereas the other 100 tuples deactivate the 100 emergencies and related 100 tacps. The experiment is repeated 100 times for a total of 20.000 tuples and 10.000 emergency activations and 10.000 emergency deactivations. The *overall activation and deactivation time* is measured varying the tuples rate, from 1.000 to 10.000 tuples per second in order to stress as much as possible the CEP system. In Figure 6.7 is shown a histogram divided into two parts, the first part represents the overall activation time, whereas the second part represents the overall deactivation time. In the activation part of the charts, the emergency instance creation time, the tacp template retrieval time and the tacp instance creation time are constant (0,5 ms, 0,6 ms,

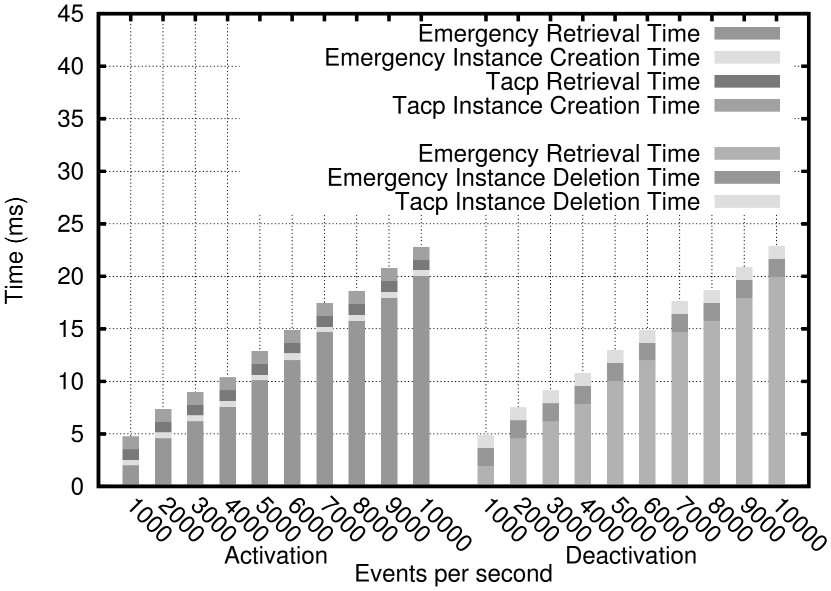


Figure 6.7: Overall Activation/Deactivation Time

0,9 ms, respectively). In the deactivation part of the chart, the emergency instance deletion time and the tacp instance deletion time are constant (1.7 ms and 2.8 ms, respectively). The emergency retrieval time grows linearly in the number of events per second (e/s) for both activation and deactivation. The tacp creation and deletion time is around 2 milliseconds for each tacp, which means in the most challenging experiment (i.e., 10.000 emergency activations and deactivations) a tacp creation/deletion overhead of 2 milliseconds over a total activation/deactivation time of 45 milliseconds, i.e., less than 5%. In this case, although the number of created/deleted tacp is high (i.e., 10.000), the overhead due to these operation is small and does not affect significantly the overall activation/deactivation time.

User Access Time

The *user access time* represents the time elapsed between the user login and the response of the system, which returns the set of objects on which the user can exercise privileges. In order to return this set, it is necessary to find the tacp/acp related to the role of the user into the tacp & acp repository (*tacp retrieval time*). Then, for each tacp, the subject, object and context conditions are checked (*tacp evaluation time*). Please note that these experiments have been conducted without considering regular access control policies, as we are mainly interested to new access control policies implied

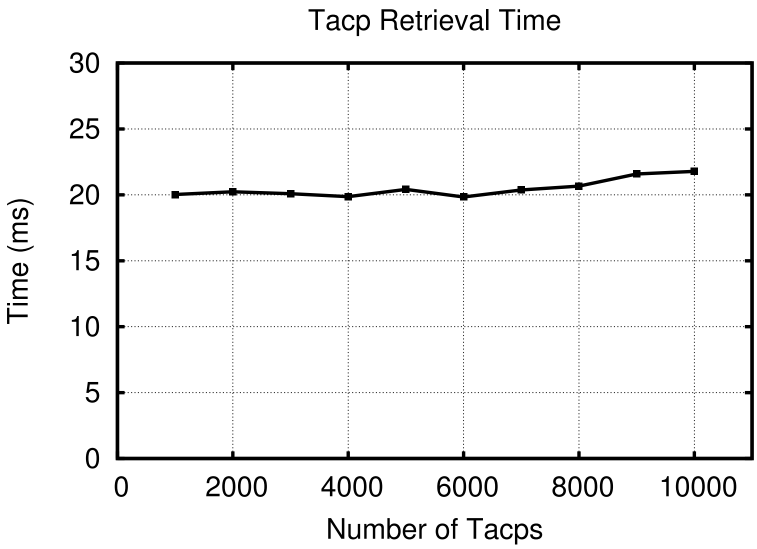


Figure 6.8: Retrieval Time of Tacp Instances

by emergency management (i.e., tacps). As such, in the following we focus on retrieval time of tacp.

Tacp Retrieval Time: in this experiment we measure the time elapsed between the user login and the arrival of the set of objects on which the user can exercise privileges. The number of tacp for each role is fixed, i.e., 100 tacps for each role, which means that each role can exercise privileges over 100 objects, assuming that each tacp identifies a unique authorized object. We carried out the experiment increasing the number of roles. More precisely, in the first experiment we make use of 10 roles, since each role can exercise privileges over 100 objects, the Access Control Handler should check 1.000 tacps. In the last experiment, we make use of 100 roles, therefore the Access Control Handler should check 10.000 tacps. The results are shown in Figure 6.8. In the first case, i.e., 1.000 tacps, the retrieval time of tacp is 20 milliseconds, while in the last case, i.e., 10.000 tacps, the retrieval time is 21,7 milliseconds. The difference between the two times is small and it guarantees a high scalability of the prototype.

Tacp Evaluation Time: in this experiment, we measure the time required to evaluate subject, object and context conditions in a single tacp. We measured this time by varying the complexity of the tacp, that is, of its subject, object and context specifications. We vary the number of conditions from 2 to 2.048 in subject, object and context specifications.

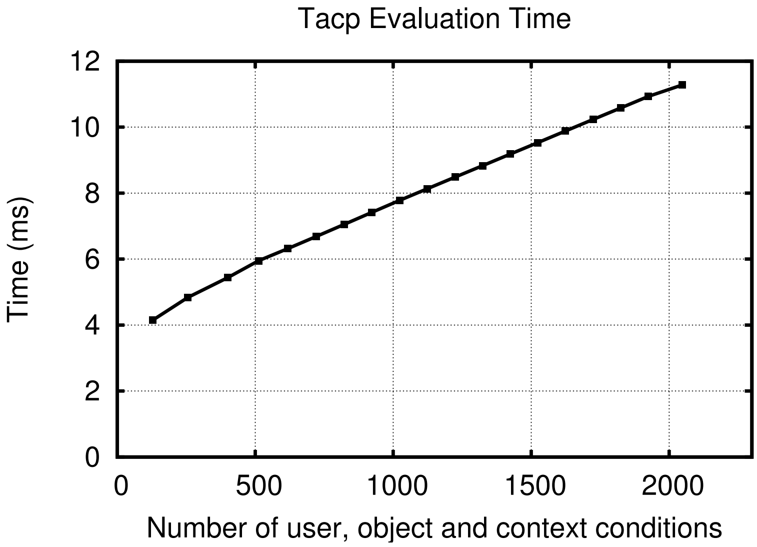


Figure 6.9: Tacp Instance Evaluation Time

In the first case, i.e. two conditions in subject, object and context, the tacp evaluation time is 3 milliseconds, while in the last case, i.e. 2.048 conditions, the time is 10,2 milliseconds. The time growth is linear in the number of conditions, as shown in Figure 6.9.

The experiment results have shown that the prototype is fast in activation/deactivation of emergency policies and more important, the access control is not affected by the emergency policy enforcement. Moreover, increasing the number of emergencies, policies and users, the time elapsed for each operation grows in a linear way, therefore the system is scalable.

6.1.3 Post Processing Time

This is the time necessary to the post processing module to check if two tuples coming out from *init* and *end* output streams of the same emergency have the same timestamp and refer to the same identifier (i.e., if a SHP hold).

In this experiment, the number of emergencies is fixed and set to 100. The emergency generator sends 100 couples of tuples (i_i, e_i) to StreamBase input streams such that i_i and e_i triggers and ends the i^{th} emergency and i_i and e_i have the same timestamp and identifier. The experiment is repeated 100 times for a total of 20.000 tuples and 10.000 SHP detections.

The *Post Processing Time* is measured varying the tuples rate, from 1.000 to 10.000 tuples per second, and, consequently, the rate of events detected by StreamBase.

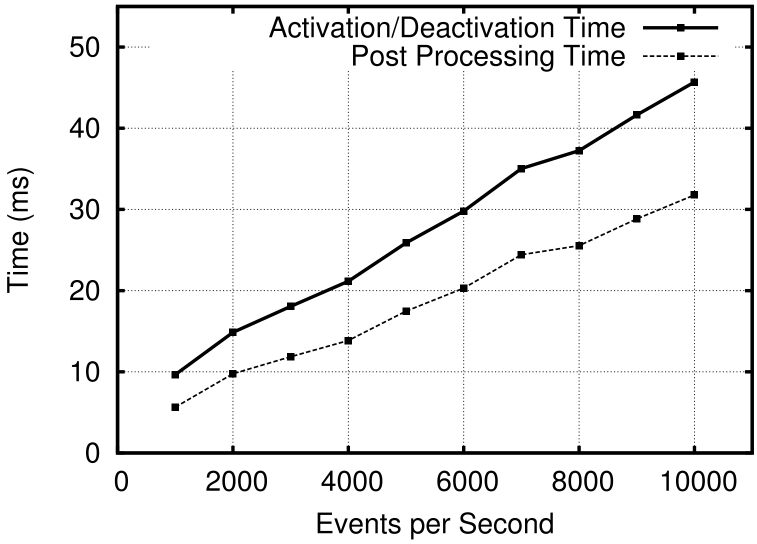


Figure 6.10: Post Processing Time vs Activation/Deactivation Time

Figure 6.10 is a comparison between post processing and activation/deactivation time. The results for emergency activation time are the same shown in Figure 3 in the paper. The chart clearly shows that post processing time is slightly better than activation/deactivation time, thus it is worth performing post processing validity checks in order to improve the overall system performance.

6.2 Unspecified Emergency Policy Evaluation

We implemented the detection of unspecified emergencies in Java on top of a StreamBase CEP platform [75] using MySQL as policies, dars and historical repository. In the following, we introduce a preliminary experiment to evaluate the effectiveness of the policy based analysis. This compares dar evaluations carried out by a group of security-aware people against dar evaluations resulting from a satisfaction-based evaluation relying on proposed measures. Then, we have carried out three experiments to evaluate satisfaction, anomaly and historical level measures separately and combined together.

6.2.1 Policy Based Analysis Evaluation

The first set of experiments we have performed aims at comparing dar evaluations carried out by a group of security-aware people against dar evaluations resulting from a satisfaction-based evaluation relying on proposed measures. The second experiment shows how evaluations change by tuning the threshold value. Both experiments have been carried on a dataset composed of 10 tacps. The dataset has been manually created so as to represent a realistic healthcare scenario. Tacps implement access requirements such as policies to authorize users with the role doctor to access patient medical records or users with the role pharmacist to access information about drugs. Tacps are defined over 47 roles and 38 object types, both hierarchically organized. Moreover, 40% of tacps contain subject or object conditions. Starting from these tacps a set of 50 dars has been generated. These dars were generated randomly, with the resulting satisfaction levels varying in a range between 0.44 and 0.94.

Satisfaction-based dar evaluation: in this experiment, we use our satisfaction level measures with a threshold value $th = 0.69$ and a tolerance value $\varepsilon = 0.05$; the threshold value has been calculated as the average of satisfaction levels of the roles-object matrix, whereas ε is an empirical value. Moreover, we set weights $w_1 = w_2 = 1$, so as to give the same importance to roles/objects and conditions. The results are reported in Table 6.1.

tot	**sl**	controlled violations	attempted abuses	ambiguous
25	$sl \leq 0.64$	0	25	0
4	$0.64 < sl < 0.74$	0	0	4
21	$sl \geq 0.74$	21	0	0

Table 6.1: Satisfaction-based dar evaluation

According to our measures, 25 dars have a low satisfaction level (i.e., $sl \leq 0.64$) therefore they should be denied since they represent attempted abuses; 4 dars have a medium satisfaction level (i.e., $0.64 < sl < 0.74$) thus they are considered ambiguous; 21 dars have a high satisfaction level (i.e., $sl \geq 0.74$), thus they should be permitted as controlled violations.

Human-based dar evaluation: each dar in the dataset has been evaluated by a group of 20 Ph.D./Master students with background in data security. Before participating to the test, all tacps in place in the system have been shown and explained to the students and the concept of controlled violation has been clearly introduced. For each dar it has been required to evaluate whether this, in his/her opinion, would have to be considered as controlled violation, attempted abuse, or the dar is ambiguous and the participant is not able to give a sure response. The participants do not know the answer given by the satisfaction-based dar evaluation, thus their decisions are not influenced by our measures results. For each dar, if more than 70% of the participants have given the same answer, this answer is considered the final human decision, whereas if the participants decisions are split (e.g., 60% controlled violations, 40% attempted abuses), the dar is considered ambiguous. The human-based evaluation results are reported in Table 6.2.

tot	**sl**	controlled violations	attempted abuses	ambiguous
25	$sl \leq 0.64$	0	25	0
4	$0.64 < sl < 0.74$	0	2	2
21	$sl \geq 0.74$	21	0	0

Table 6.2: Human-based dar evaluation

The human-based evaluation suggests denying the 25 dars with lowest level of satisfaction, since they represent attempted abuses; for the 4 dars in the middle, the participants recognized 2 dars as attempted abuses and they split on the other 2 dars; for the last 21 dars, they recognized all of them as controlled violations.

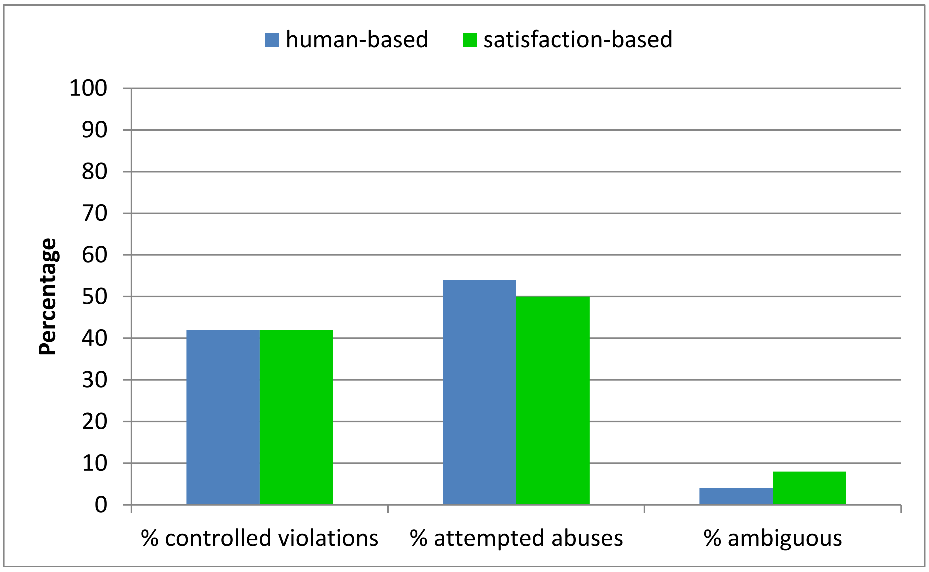


Figure 6.11: Human-based vs. Satisfaction-based Dars Evaluation

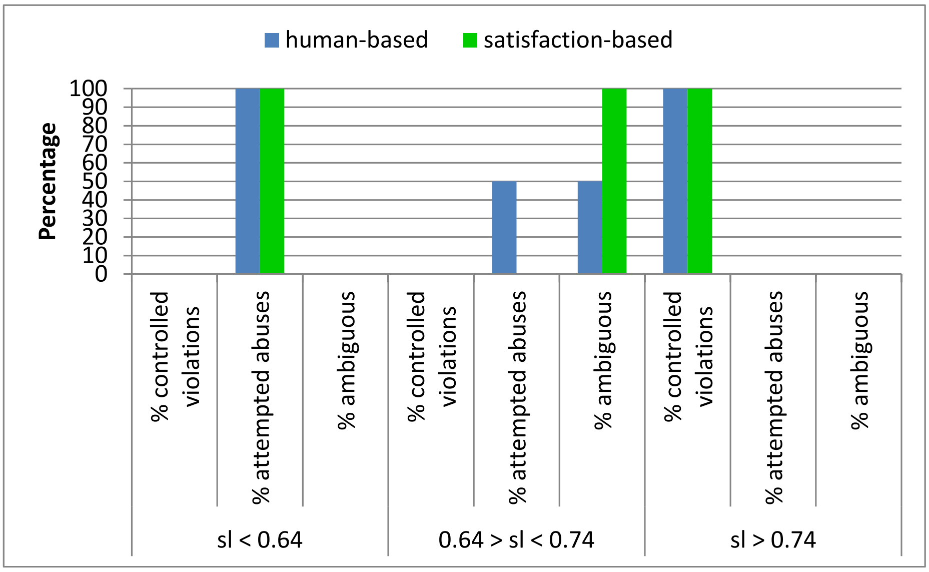


Figure 6.12: Grouped Human-based vs. Satisfaction-based Dars Evaluation

A comparison between human-based and satisfaction-based evaluation is given in Figure 6.11, which shows that exactly the same percentage of dars (i.e., 42%) have been judged as controlled violations by both evaluations. Regarding dars judged as attempted abuses, human-based evaluation is more restrictive since participants have judged 54% of the dars, while our measures only 50%. Moreover, participants have considered 4% of the dars ambiguous, whereas satisfaction-based evaluation has recognized 8% of the dars as ambiguous. Indeed, there is a 4% of dars which are considered ambiguous by satisfaction-based evaluation, but judged as attempted abuse by participants of our evaluation.

A more specific analysis is given in Figure 6.12, where human and satisfaction-based evaluations are grouped by satisfaction levels. The histogram highlights that for low (i.e., $sl \leq 0.64$) and high satisfaction levels (i.e., $sl \geq 0.74$) the two evaluations match. They are different for dars with medium satisfaction level (i.e., $0.64 < sl < 0.74$). In this case, satisfaction-based evaluation considers 100% of the dars ambiguous, whereas participants recognized 50% of dars to be denied and 50% as ambiguous.

These results highlight that the proposed measures correctly recognize dars which are close to satisfy existing tacps and dars which are distant. Regarding ambiguous dars, the satisfaction-based decision correctly recognizes those dars which are tagged as ambiguous by humans, thus there are no dars allowed by the system, but ambiguous for human beings. However there is a small percentage (4% of the total dars and 50% of ambiguous dars), which are recognized as ambiguous by satisfaction-based evaluation, but are denied by human participants. We believe that this percentage can be reduced by properly setting the threshold and tolerance values, moreover it is possible, for ambiguous dars, to let the user decide (taking the responsibility of this action) whether to access the data or not.

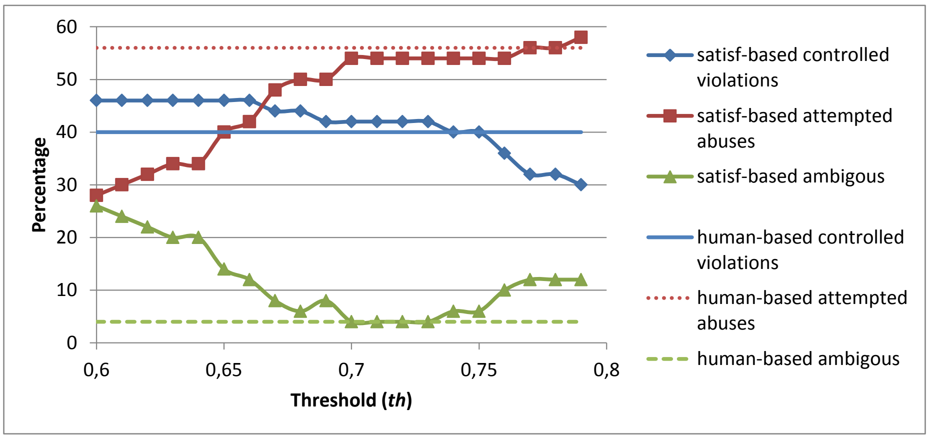


Figure 6.13: Human-based vs Satisfaction-based Dars Evaluation varying threshold value

6.2.2 Access Requests Analysis Evaluation

In the following, we introduce the dataset used for the experiments. Based on the proposed dataset, we have carried out three experiments to evaluate our measures separately, i.e., only the satisfaction level measure, only the anomaly level measure and only the historical level measure. Then, we have run the same experiment combing the three measures using a simple average and a weighted average. In addition, we carried out another experiment varying the threshold value, in order to check how this value influence the detection of unspecified emergencies. Finally, based on these results we analyze possible methods to combine our measures in order to have the best detection and management of unspecified emergencies.

6.2.3 Dataset

The dataset is modeled among two different domains: healthcare and temperature. Regarding healthcare domain, the dataset contains health measures taken from patients wearing several monitoring devices that catch their vital signs. These dataset includes real data coming from an ECG dataset [55] and synthetic data created *ad hoc* for our experiments. In temperature domain, the dataset contains sensor data generated measuring temperature in the Intel Berkeley Research Laboratory [41].

We model these two dataset into two different streams called `VitalSigns` for the healthcare domain and `Temperature` for the temperature domain. The `VitalSigns` stream contains 7 attributes, i.e., heart rate, temperature, systolic and diastolic pressure, glucose level, respiratory rate and patient, whereas the `Temperature` stream attributes are date, time, room, building and temperature.

During the experiment, an emergency generator sends to the two streams a large amount of tuples containing regular values and few tuples containing anomalous values which signal emergency situations. More precisely, for the `VitalSigns` stream, the generator sends 23.000 tuples. Among these tuples there are 19 small sets of tuples containing anomalous values. These sets have an average size of 50 tuples. Among these anomalies 9 are defined over the heart rate attribute, 2 over temperature, 2 over glucose rate, 2 over systolic pressure, 2 over diastolic pressure, 2 over respiratory rate.

For the `Temperature` stream, the generator sends 2 millions of tuples; among these tuples there are 6 sets of anomalous tuples. These sets have an average size of 60 tuples and the anomalies are all defined over the temperature attribute. The total amount of sets of anomalous tuples for both streams is 25.

Starting from the 25 sets of anomalous tuples 25 emergencies have been modeled. For instance, the regular value for the heart rate attribute is around 80 bpm; consider a set of tuples containing values around 50 bpm. These tuples represent an anomaly thus, we can model a bradycardia emergency when the heart rate value become lower than 60 bpm. Once we have defined these 25 emergencies, we associated them with 25 tacps, one for each emergency. Tacps implement access requirements such as policies to authorize users with the role doctor to access patient medical records or users with the role pharmacist to access information about drugs. Tacps are defined over 47 roles and 38 object types, both hierarchically organized. Moreover, 40% of tacps contain subject or object conditions.

Besides the tuples dataset and the emergency policies repository, we also created a dataset of 50 access requests. Before running the experiment we label these access request as “To Deny” (TD) or “To Permit” (TP). More precisely, we created 25 access requests which satisfy existing tacps that are labeled as TP and 25 access requests which not satisfy existing tacps which are labeled as TD.

During the experiment, an emergency generator sends tuples belonging to the tuple dataset in order to trigger the 25 emergencies. Every time an emergency tuple is sent to the CEP platform the related emergency is triggered and the related tacp is activated, thus the emergency generator sends an access request for the object protected by the tacp. This is repeated until all 25 access requests labeled as TP have been sent. Once all access requests in TP have been sent, the other 25 access requests labeled as TD are also sent to the system.

The first time we run the experiment, all the 25 emergency policies are active, thus the 25 correlated emergencies are detected and the 25 access requests labeled as TP are permitted, whereas the other 25 access requests labeled as TD are denied. Then, we randomly and incrementally hide part of emergency policies in order to verify if hidden emergencies are still detected as anomalies and hidden tacps are still permitted as controlled violations. The experiment is repeated three times: (1) using only the satisfaction level measure (2) using only the anomaly level measure and (3) using only the historical level measure. In the end, we also run the same experiment combing the three measures using a simple average and a weighted average. In addition, we run another experiment varying the threshold value, in order to check how this value influence the performance of our measures.

6.2.4 Satisfaction Level Evaluation

In this experiment, the emergency generator sends tuples from the tuples dataset and access requests from the access request dataset. During the experiment, we randomly and incrementally hide emergency policies in order to verify if hidden policies are permitted as controlled violations exploiting our satisfaction level measure. We repeat the experiment 10 times hiding two policies every time. The results are reported in Table 6.3, which shows the percentage of Access Requests (AR) labeled as TP that are effectively permitted and the percentage of ARs labeled as TD that are effectively denied. Table 6.3 shows also the percentage of False Negatives (FNs), i.e., access requests that are denied even though they should be permitted and the percentage of False Positives (FPs), i.e., access requests that are permitted even though they should be denied.

The results in Table 6.3 shows that when the number of hidden emergency policies is low, i.e., lower than 40%, all the access requests are evaluated correctly, i.e., 100% of access requests to be permitted are effectively permitted by satisfaction levels and 100% of access requests to be denied are effectively denied. Indeed, when there are too many hidden emergency policies (i.e.,

% of hidden EP	**% of permitted ARs**	**% of denied ARs**	**% FNs**	**% FPs**
8	100	100	0	0
16	100	100	0	0
24	100	100	0	0
32	100	100	0	0
40	100	100	4	0
48	98	100	2	0
56	88	100	12	0
64	78	100	22	0
72	70	100	30	0
80	60	100	40	0

Table 6.3: Satisfaction Level Evaluation

more than 48%), the satisfaction level is less precise and some FNs occur.[1]

6.2.5 Anomaly Level Evaluation

This is the same of the previous experiment where we sends tuples from the tuples dataset, access requests from the access request dataset and randomly and incrementally hide emergency policies (two policies every time), but in this case we exploit anomaly level measure. During the experiment, the generic anomaly detection function *adf*() presented in Section 4.3, is replaced by two different techniques [56, 11] whose details are explained in Section 2. The results are reported in Table 6.4.

% of hidden emergency policies	**% of permitted AR**	**% of denied AR**	**% FNs**	**% FPs**
8	100	100	0	0
16	100	100	0	0
24	100	100	0	0
32	100	100	0	0
40	100	100	0	0
48	96	98	4	2
56	96	96	4	4
64	96	92	4	8
72	96	82	4	18
80	96	74	4	26

Table 6.4: Anomaly Level Evaluation

[1] FPs are all zeroes, in this case, because when the number of emergency policies decreases it is not possible to have satisfaction levels that increase.

The results in Table 6.4 shows, as in the previous experiment that when the number of hidden emergency policies is low, i.e., lower than 40%, then 100% of access requests are evaluated correctly. When there are too many hidden emergency (i.e., more than 48%) the anomaly level is less precise and there are 10% of FNs. In this case, there are also some FPs. This happens when the number of hidden policies is greater than 48%, because in this case the anomaly level is less precise, thus some normal events are detected as emergencies and the related access requests are permitted even though they should be denied.

6.2.6 Historical Level Evaluation

This experiment is quite different from the previous ones. The emergency generator sends tuples and access requests selected from the corresponding datasets. During the experiment, we randomly and incrementally hide emergency policies. Every time we hide emergency policies some of these policies are permitted as controlled violations[2] populating in this way the historical repository. We repeat the experiment 12 times hiding two emergency policy every time. The results are reported in Table 6.5.

% of hidden tacps	**number of controlled violations**	**% of permitted AR**	**% of denied AR**	**% FNs**	**% FPs**
8	0	0	100	100	0
16	2	0	100	100	0
24	4	0	100	100	0
32	6	36	100	64	0
40	8	50	100	50	0
48	10	60	100	40	0
56	12	72	100	38	0
64	14	76	100	34	0
72	16	84	100	16	0
80	18	86	100	14	0
100	20	92	100	8	0
100	22	100	100	0	0

Table 6.5: Historical Level Evaluation

The results in Table 6.5 shows that performances of historical level evaluation are slightly worse than performances of other measures. In the first row, we hide 8% of emergency policies, but the historical repository is empty

[2] In this case, we use both satisfaction levels and historical level; satisfaction level is used to populate the historical repository and historical level is used for the evaluation.

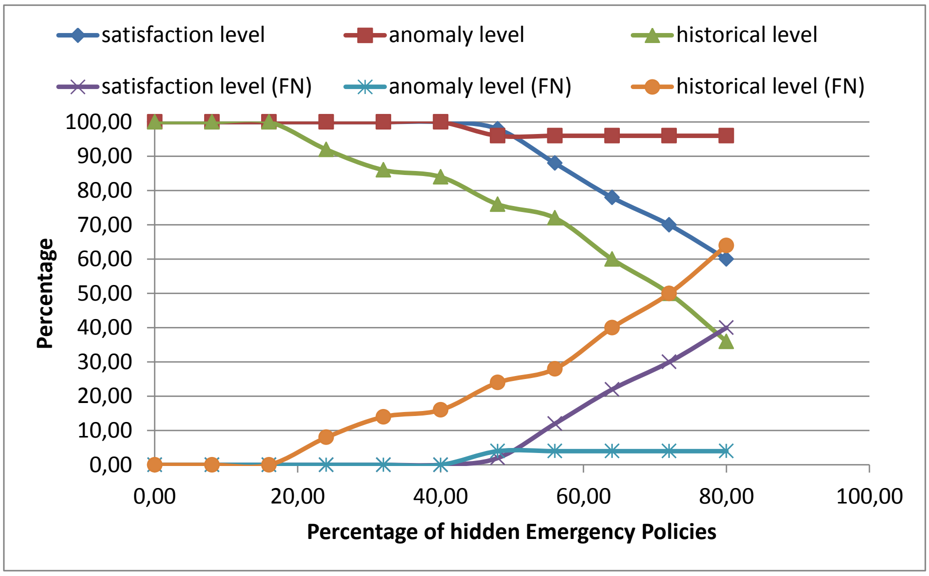


Figure 6.14: Access Requests Permitted as Controlled Violations

(i.e., zero controlled violations), thus all access requests are denied, i.e., 100% of FNs. While the number of controlled violations in the historical repository increases the percentage of permitted access requests increases as well and the percentage of FNs decreases.[3]

6.2.7 Satisfaction, Anomaly and Historical Level Comparison

In this section, we analyze the results of previous experiments comparing performances of satisfaction, anomaly and historical level from the point of view of access requests permitted and access request denied.

The comparison from the point of view of access requests permitted as controlled violations is shown is Figure 6.14.

As shown in Figure 6.14 satisfaction and anomaly level permit 100% of access requests that should be permitted when the percentage of hidden emergency policies is lower than 40%, whereas historical level permits 100% of access requests when the percentage of hidden controlled violations is lower than 16%. Indeed, performance of historical level is lower than performance of other measures. When the number of hidden emergency policies increases

[3] FPs are all zeros in this case for the same reasons explained in the satisfaction level experiment.

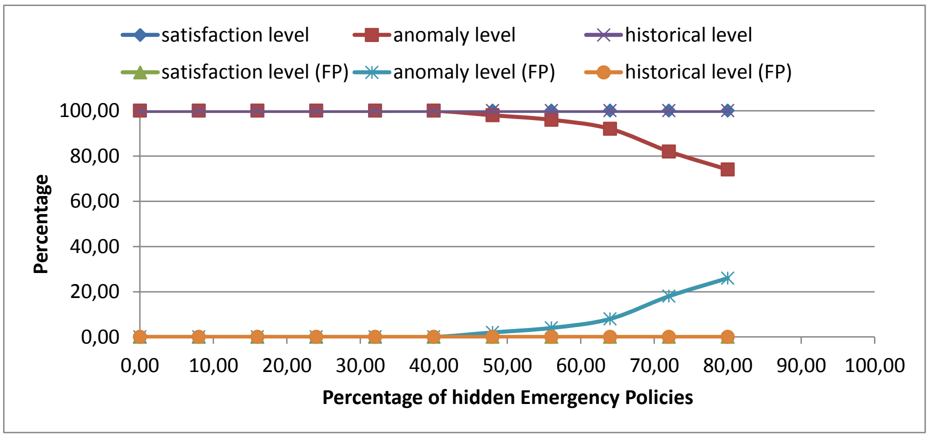


Figure 6.15: False Negatives Comparison

the percentage of permitted access requests decrease in a lower manner for satisfaction level and anomaly level. Conversely, the number of FNs increases when the number of hidden emergency policies increases. The number of FNs increases in a faster manner for historical level since performance of historical measure is lower.

The comparison from the point of view of denied access requests is shown is Figure 6.15.

As shown in Figure 6.15 satisfaction and historical level deny 100% of access requests that should be denied, whereas anomaly level deny 100% of access requests when the percentage of hidden controlled violations is lower than 48%. Indeed, satisfaction and historical level do not have any FPs, while for anomaly level the number of FPs increases when the number of hidden emergency policies increases.

6.2.8 Threshold Evaluation

In this section, we evaluate the importance of threshold selection. In order to perform this test, we decide to hide 40% of emergency policies. Since for higher percentages of hidden policies the performance of our measures decay, we decided to hide 40% because this is the maximum number of hidden policies for which our strategies work efficiently. We vary the threshold value from 0.3 to 0.75 and the results are shown in Figure 6.16.

As shown in Figure 6.16, our measures have the best performance with the threshold set to 0.6, whereas when the threshold changes the number of FPs and FNs decreases or increases. More precisely, when the threshold is

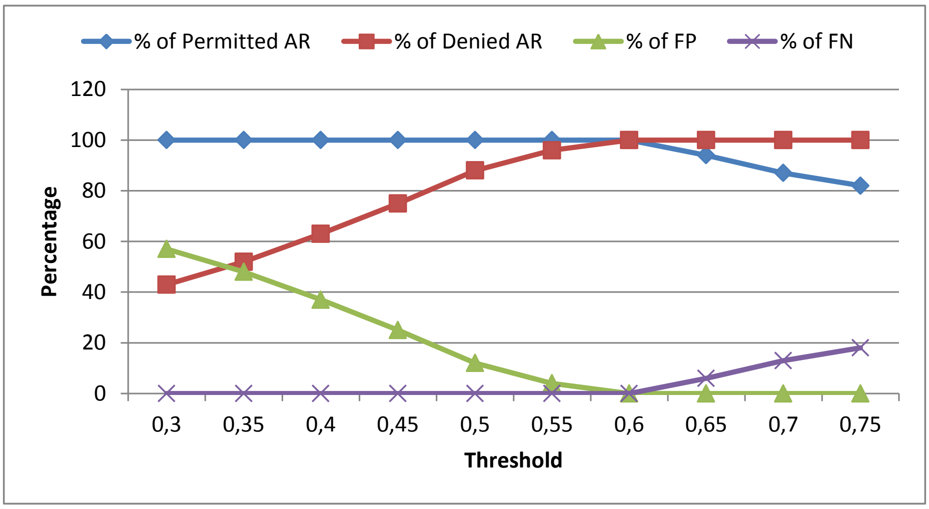


Figure 6.16: Threshold Evaluation

lower than 0.6, the system authorizes too many access requests (i.e., FPs), whereas when the threshold is higher than 0.6, the system deny too many access requests (i.e., FNs).

6.2.9 Experiments Discussion

In this section, we propose three different way of combining our emergency detection strategies, based on the results of our experiments. The first two ways are the *average* and the *weighted average*. We carried out an experiment to evaluate these two strategies. This is the same of the previous experiment where we randomly and incrementally hide emergency policies (two policies every time), but in this case we combine our measures using average (avg) of satisfaction, anomaly and historical levels and weighted average ($wavg$) where historical level is calculated as shown in Equation 6.1, i.e., the dar-similarity has more weight than the event set similarity level.

$$hl(t,d) = \frac{0.7 \times dar - sim(d_1, d_2) + 0.3 \times essl(E_{a_1}, E_{a_2})}{2} \tag{6.1}$$

Equation 6.1 calculates historical level as the weighted average between the dars similarity level *dar-sim* (Figure 4.4) and the event set similarity level *essl* (Definition 4.4.6). The results of the comparison between *avg* and *wavg* are reported in Figures 6.17 and 6.18. In Figure 6.17, the comparison is shown from the point of view of permitted access requests, whereas in Figure 6.18 from the point of view of denied access requests.

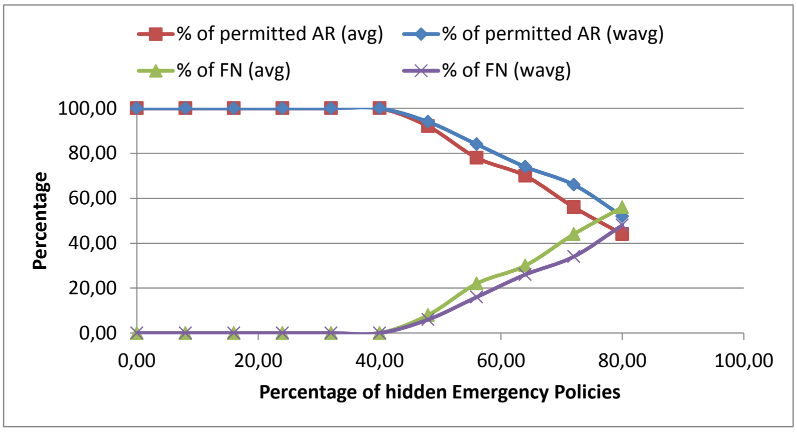


Figure 6.17: Avg vs Weighted Avg (Permitted AR)

As shown in Figure 6.17, *wavg* and *avg* combinations work in the same way, i.e., 100% of access requests that should be permitted are effectively permitted when the percentage of hidden emergency policies is lower than 40%, whereas when the number of hidden emergency policies increases *wavg* is slightly better than *avg* for both number of permitted access requests and number of FNs.

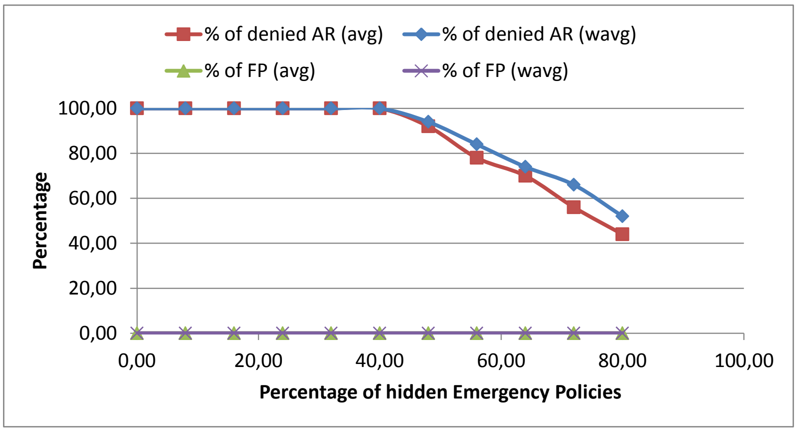


Figure 6.18: Avg vs Weighted Avg (Denied AR)

As shown in Figure 6.18, *wavg* and *avg* combinations work in the same way, i.e., 100% of access requests that should be denied are effectively denied when the percentage of hidden emergency policies is lower than 40%, whereas when the number of hidden emergency policies increases *wavg* is slightly better than *avg* in the number of denied access requests. The number of FPs is zero because when the number of emergency policies decreases it is not likely that our measures increase.

Although the performance of the average are good a weighted average might be better depending of the domain and the base knowledge. For instance, when the number of emergency policies stored in the policy repository is large, the accuracy of satisfaction level increases, thus it is better to assign a greater weight to these measure. Moreover, weights assigned to the measures might change over time. For instance, in the beginning, when the historical repository is empty, it is better to assign a lower weight to the historical level, but when the number of controlled violations increases, the weight of historical level might be increased as well, since the base knowledge is larger.

Another way of combining these measures might be a sequential evaluation of satisfaction levels. The evaluation might be predefined by the system administrator or it might be decided at runtime. In the former case, the administrator defines a-priori the depth of the evaluation, e.g., depth is set to 1, 2 or 3, thus when an access request is denied, if depth is set to 1, only the satisfaction level is calculated, if depth is set to 2 a combination of satisfaction and anomaly level is calculated, otherwise all the three measures are combined. In the latter case, the system decide at runtime which strategy to apply, i.e., when an access request is denied the satisfaction level is calculated, if this level clearly identifies whether the access request is to authorize or not, the result is returned, otherwise the measure is refined with the anomaly level, in this case again if this level clearly identifies whether to authorize or not the access request, the result is returned, otherwise the measure is refined with the historical level. In addition, this way of evaluate the access request can be refined assigning different weights to the three levels at runtime.

The number of possible combinations of our strategies is large, thus it is not possible to general a general way to decide which combination is better. This decision is strictly dependent on the domain of the system and the requirements of access control and information sharing.

Chapter 7

Conclusions

The general goal of this PhD work concerned the definition, implementation and testing of an access control framework to enforce controlled information sharing in emergency situations.

Traditional access control systems do not fit the emergency management scenario, where there is the need for a more efficient, timely and flexible information sharing. For these reasons, we have proposed a novel access control model based on emergency policies. The *core* model is able to express and detect complex emergency situations exploiting CEP technologies. Moreover, we extend regular access control policies with *temporary access control policies* that override regular policies during emergency situations. Such policies also support obligations, i.e., set of actions that must be fulfilled when a certain event occurs in the system.

The *core emergency policy* model has been extended in order to support *composed emergencies* and *administration policies*. The emergency policy composition introduces the concept of composed emergencies to describe how atomic emergencies can be combined together to from a composed one and how sub-emergencies can be overridden by a composed one. The emergency policy administration is enforced defining proper scopes for emergency policies that limit the right to state emergency policies only to specific emergencies.

In this section, we introduce our framework to extend the core model in order to deal with unspecified emergencies. The management of unspecified emergencies is performed using three different strategies: (1) *policy based analysis* (2) *anomaly based analysis* and (3) *historical based analysis*. The *policy based analysis* calculates how much an access request is close to satisfy existing policies; if an access request is significantly close to satisfy a tacp, it is likely that the access requests represents an information need related to an unspecified emergency. The *anomaly based analysis* finds anomalies related to a denied access request; if an anomaly is correlated to an access request, it is likely that the anomaly represents an unspecified emergency and the access requests represents the related emergency policy. The *historical based analysis* considers previously permitted access requests in order to detect if the current access request is similar to one of them; indeed if this is true the access requests should be authorized. The architecture of our framework is shown in Figure 4.1 which illustrates also how the system enforces the three strategies for the management of unspecified emergencies.

In order to deal with *unspecified emergencies*, the core emergency policy model has been further extended to support flexible information sharing also for unplanned emergency situations. The basic idea to manage these situations is to detect unspecified emergencies exploiting anomaly detection techniques and to permit those access requests that are denied due to the absence of policies related to unspecified emergencies. Obviously, not all access request related to an emergency should be allowed, but only those access requests related to unspecified emergencies. In order to detect whether an access request is related to an unspecified emergency or it is an attempted abuse, we have defined three different strategies called policy, anomaly and historical based analysis based on three different satisfaction level measures.

In addition to the proposed access control model, we have implemented a prototype framework called SHARE (Secure information sHaring frAmework for emeRgency managemEnt) and we have carried out an extensive set of tests in order to check what is the impact of emergency policies into an access control system and to evaluate the effectiveness of the proposed techniques for the management of unspecified emergencies.

The experiment results of the core access control model have shown that the prototype is fast in activation/deactivation of emergency policies and more important, the access control is not affected by the emergency policy enforcement. Moreover, increasing the number of emergencies, policies and users, the time elapsed for each operation grows in a linear way, therefore the system is scalable.

Regarding the management of unspecified emergencies, results of experiments for the satisfaction level measures shows that the proposed techniques correctly recognize anomalous events which signals emergency situations, access requests which are close to satisfy existing policies, i.e., to be permitted as controlled violations and also access requests distant from existing policies, i.e., to be denied as attempted abuses. Moreover, experiment results highlight that with an accurate tuning of threshold values the percentage of errors is irrelevant.

The work presented in this report might be extend along several directions.

- Regarding specified emergencies, we believe that tools to assist security administrator in emergencies and emergency policies definitions can be defined. More precisely, since a large number of risk assessment tools have been developed in the last years [66], we plan to analyze them so as to automatically extract emergency descriptions, policies and obligations from emergency scenarios and response plans.

- Regarding unspecified emergencies, we would like to extend the approach to take into account user trust, object confidentiality levels and *a-posteriori* log analysis. Moreover, We plan to extend our approach to the support of unspecified emergencies that are not similar to any of the registered emergencies. We also plan to develop learning techniques to automatically define new emergency policies based on occurred dars and controlled violations.

- We aim to enforce information sharing among multiple organizations exploiting new cloud computing techniques. Cloud computing is suitable for the purpose of information sharing because it provides a common storage space where organizations can share their data. Starting from works in the field of collaboration in disaster management, we believe we could use a cloud infrastructure to solve most of the interoperability issues due to heterogeneous access control models.

Bibliography

[1] *The 9/11 commission report*, tech. rep., National Commission on Terrorist Attacks Upon the United States, July 2004.

[2] *Federal response to hurricane katrina: Lessons learned*, tech. rep., Assistant to the President for Homeland Security and Counter Terrorism, February 2006.

[3] A. Abu Safia and Z. Al Aghbari, *Searching data streams for variable length anomalies*, in Innovations in Information Technology (IIT), 2011 International Conference on, 2011, pp. 297–302.

[4] A. Adi and O. Etzion, *Amit - the situation manager*, The VLDB Journal, 13 (2004), pp. 177–203.

[5] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman, *Efficient pattern matching over event streams*, in Proceedings of the 2008 ACM SIGMOD international conference on Management of data, SIGMOD '08, New York, NY, USA, 2008, ACM, pp. 147–160.

[6] K. Alghathbar and D. Wijesekera, *Consistent and complete access control policies in use cases*, in UML 2003 - The Unified Modeling Language. Modeling Languages and Applications, P. Stevens, J. Whittle, and G. Booch, eds., vol. 2863 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2003, pp. 373–387.

[7] Y. Amano, *Statement to iaea ministerial conference on nuclear safety*, 2011.

[8] C. Ardagna, S. De Capitani di Vimercati, S. Foresti, T. Grandison, S. Jajodia, and P. Samarati, *Access control for smarter healthcare using policy spaces*, Computers and Security, 29 (2010), pp. 848–858.

[9] S. Barker, *The next 700 access control models or a unifying meta-model?*, in Proceedings of the 14th ACM Symposium on Access Control Models and Technologies, SACMAT '09, New York, NY, USA, 2009, ACM, pp. 187–196.

[10] J. Bauckmann, U. Leser, F. Naumann, and V. Tietz, *Efficiently detecting inclusion dependencies*, in In Int. Conf. on Data Engineering (ICDE 07, Poster, 2007.

[11] M. S. Beigi, S.-F. Chang, S. Ebadollahi, and D. C. Verma, *Anomaly detection in information streams without prior domain knowledge*, IBM J. Res. Dev., 55 (2011), pp. 550–560.

[12] E. Bertino, P. A. Bonatti, and E. Ferrari, *Trbac: A temporal role-based access control model*, ACM Trans. Inf. Syst. Secur., 4 (2001), pp. 191–233.

[13] E. Bertino, C. Brodie, S. B. Calo, L. F. Cranor, C. Karat, J. Karat, N. Li, D. Lin, J. Lobo, Q. Ni, P. R. Rao, and X. Wang, *Analysis of privacy and security policies*, IBM J. Res. Dev., 53 (2009), pp. 225–241.

[14] C. Bertolissi and M. Fernández, *A rewriting framework for the composition of access control policies*, in Proceedings of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming, PPDP '08, New York, NY, USA, 2008, ACM, pp. 217–225.

[15] ——, *Category-based authorisation models: Operational semantics and expressive power*, in Proceedings of the Second International Conference on Engineering Secure Software and Systems, ESSoS'10, Berlin, Heidelberg, 2010, Springer-Verlag, pp. 140–156.

[16] C. Bettini, S. Jajodia, X. Wang, and D. Wijesekera, *Obligation monitoring in policy management*, in Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY'02), POLICY '02, Washington, DC, USA, 2002, IEEE Computer Society, pp. 2–.

[17] C. Bettini, S. Jajodia, X. S. Wang, and D. Wijesekera, *Provisions and obligations in policy management and security applications*, in Proceedings of the 28th international conference on Very Large Data Bases, VLDB '02, VLDB Endowment, 2002, pp. 502–513.

[18] H. L. Bill Parducci, *extensible access control markup language (xacml) specification 3.0*, August 2010.

[19] W. N. Blog, *Emergency responders take 911 calls side by side*, 2012.

[20] P. Bonatti, S. De Capitani di Vimercati, and P. Samarati, *An algebra for composing access control policies*, ACM Trans. Inf. Syst. Secur., 5 (2002), pp. 1–35.

[21] A. Boukottaya and C. Vanoirbeek, *Schema matching for transforming structured documents*, in Proceedings of the 2005 ACM symposium on Document engineering, DocEng '05, New York, NY, USA, 2005, pp. 101–110.

[22] A. D. Brucker and H. Petritsch, *Extending access control models with break-glass*, in Proceedings of the 14th ACM symposium on Access control models and technologies, SACMAT '09, New York, NY, USA, 2009, ACM, pp. 197–206.

[23] A. D. Brucker, H. Petritsch, and S. G.Weber, *Attribute-based encryption with break-glass*, in Workshop In Information Security Theory And Practice (WISTP), P. Samarati, M. Tunstall, and J. Posegga, eds., no. 6033 in Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, 2010, pp. 237–244.

[24] G. Bruns, D. S. Dantas, and M. Huth, *A simple and expressive semantic framework for policy composition in access control*, in Proceedings of the 2007 ACM workshop on Formal methods in security engineering, FMSE '07, New York, NY, USA, 2007, ACM, pp. 12–21.

[25] G. Bruns and M. Huth, *Access control via belnap logic: Intuitive, expressive, and analyzable policy composition*, ACM Trans. Inf. Syst. Secur., 14 (2011), pp. 9:1–9:27.

[26] B. Carminati, E. Ferrari, and M. Guglielmi, *Secure information sharing for specified and unspecified emergencies*, Under Submission to Data & Knowledge Engineering.

[27] ——, *Secure information sharing on support of emergency management*, in Privacy, Security, Risk and Trust (PASSAT), 2011 IEEE Third International Conference on and 2011 IEEE Third International Conference on Social Computing (SocialCom), oct. 2011, pp. 988–995.

[28] ——, *Policies for composed emergencies in support of disaster management*, in Secure Data Management (SDM), W. Jonker and M. Petković, eds., vol. 7482 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 75–92.

[29] ——, *Controlled information sharing for unspecified emergencies*, in In proceedings of the 8th International Conference on Risks and Security of Internet and Systems (CRISIS), Oct. 2013.

[30] ——, *Share: Secure information sharing framework for emergency management*, in Data Engineering (ICDE), 2013 IEEE 29th International Conference on, 2013, pp. 1336–1339.

[31] ——, *A system for timely and controlled information sharing in emergency situations*, IEEE Transactions on Dependable and Secure Computing (TDSC), 10 (2013), pp. 129–142.

[32] V. Chandola, A. Banerjee, and V. Kumar, *Outlier detection: A survey*, 2007.

[33] V. Chandola, A. Banerjee, and V. Kumar, *Anomaly detection: A survey*, ACM Comput. Surv., 41 (2009), pp. 15:1–15:58.

[34] E.-A. Cho, G. Ghinita, and E. Bertino, *Privacy-preserving similarity measurement for access control policies*, in Proceedings of the 6th ACM workshop on Digital identity management, DIM '10, New York, NY, USA, 2010, pp. 3–12.

[35] M. J. Covington, W. Long, S. Srinivasan, A. K. Dev, M. Ahamad, and G. D. Abowd, *Securing context-aware applications using environment roles*, in Proceedings of the sixth ACM symposium on Access control models and technologies, SACMAT '01, New York, NY, USA, 2001, ACM, pp. 10–20.

[36] J. Crampton and G. Loizou, *Administrative scope: A foundation for role-based administrative models*, ACM Trans. Inf. Syst. Secur., 6 (2003), pp. 201–231.

[37] G. Cugola and A. Margara, *Tesla: a formally defined event specification language*, in DEBS, 2010, pp. 50–61.

[38] M. L. Damiani, E. Bertino, B. Catania, and P. Perlasca, *Georbac: A spatially aware rbac*, ACM Trans. Inf. Syst. Secur., 10 (2007).

[39] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, *The ponder policy specification language*, in Proceedings of the International Workshop on Policies for Distributed Systems and Networks, POLICY '01, London, UK, 2001, Springer-Verlag, pp. 18–38.

[40] G. Ding, H. Dong, and G. Wang, *Appearance-order-based schema matching*, in Proceedings of the 17th international conference on Database Systems for Advanced Applications - Volume Part I, DASFAA'12, Berlin, Heidelberg, 2012, pp. 79–94.

[41] I. R. Division, *Temperature dataset*, Sept. 2013. http://db.csail.mit.edu/labdata/.

[42] P. H. dos Santos Teixeira and R. L. Milidiú, *Data stream anomaly detection through principal subspace tracking*, in Proceedings of the 2010 ACM Symposium on Applied Computing, SAC '10, New York, NY, USA, 2010, ACM, pp. 1609–1616.

[43] M. Eckert and F. Bry, *Rule-based composite event queries: the language xchangeeq and its semantics*, Knowl. Inf. Syst., 25 (2010), pp. 551–573.

[44] M. Elahi, K. Li, W. Nisar, X. Lv, and H. Wang, *Efficient clustering-based outlier detection algorithm for dynamic data stream*, in Fuzzy Systems and Knowledge Discovery, 2008. FSKD '08. Fifth International Conference on, vol. 5, 2008, pp. 298–304.

[45] E. F. Elisa Bertino, Silvana Castano and M. Mesiti, *Specifying and enforcing access control policies for xml document sources*, World Wide Web, 3 (2000), pp. 139–151. 10.1023/A:1019289831564.

[46] J. Evermann, *Theories of meaning in schema matching: An exploratory study*, Inf. Syst., 34 (2009), pp. 28–44.

[47] T. F. E. M. A. (FEMA), *Emergency response plan implementation @ONLINE*, Sept. 2012.

[48] E. Ferrari, *Access control in data management systems*, Synthesis Lectures on Data Management, 2 (2010), pp. 1–117.

[49] A. Ferreira, D. Chadwick, P. Farinha, R. Correia, G. Zao, R. Chilro, and L. Antunes, *How to securely break into rbac: The btg-rbac model*, in Proceedings of the 2009 Annual Computer Security Applications Conference, ACSAC '09, Washington, DC, USA, 2009, IEEE Computer Society, pp. 23–31.

[50] A. Ferreira, R. Cruz-Correia, L. Antunes, P. Farinha, E. Oliveira-Palhares, D. W. Chadwick, and A. Costa-Pereira, *How to break access control in a controlled manner*, in Proceedings of the 19th IEEE Symposium on Computer-Based Medical Systems, Washington, DC, USA, 2006, IEEE Computer Society, pp. 847–854.

[51] D. Gyllstrom, E. Wu, H. Chae, Y. Diao, P. Stahlberg, and G. Anderson, *Sase: Complex event processing over streams*, in In Proceedings of the Third Biennial Conference on Innovative Data Systems Research, 2007.

[52] K. Irwin, T. Yu, and W. H. Winsborough, *Assigning responsibility for failed obligations.*

[53] K. Irwin, T. Yu, and W. H. Winsborough, *On the modeling and analysis of obligations*, in Proceedings of the 13th ACM conference on Computer and communications security, CCS '06, New York, NY, USA, 2006, ACM, pp. 134–143.

[54] R. Jiang, H. Fei, and J. Huan, *Anomaly localization for network data streams with graph joint sparse pca*, in Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining, KDD '11, New York, NY, USA, 2011, ACM, pp. 886–894.

[55] E. Keogh, L. Jessica, and F. Ada, *ECG dataset*, Sept. 2013. http://www.cs.ucr.edu/∼eamonn/discords/.

[56] E. Keogh, J. Lin, and A. Fu, *Hot sax: Efficiently finding the most unusual time series subsequence*, in Proceedings of the Fifth IEEE International Conference on Data Mining, ICDM '05, Washington, DC, USA, 2005, IEEE Computer Society, pp. 226–233.

[57] D. R. Kuhn, E. J. Coyne, and T. R. Weil, *Adding attributes to role-based access control*, Computer, 43 (2010), pp. 79 –81.

[58] A. J. Lee, J. P. Boyer, L. E. Olson, and C. A. Gunter, *Defeasible security policy composition for web services*, in Proceedings of the fourth ACM workshop on Formal methods in security, FMSE '06, New York, NY, USA, 2006, ACM, pp. 45–54.

[59] N. Li and Z. Mao, *Administration in role-based access control*, in Proceedings of the 2nd ACM symposium on Information, computer and

communications security, ASIACCS '07, New York, NY, USA, 2007, ACM, pp. 127–138.

[60] D. Lin, P. Rao, E. Bertino, N. Li, and J. Lobo, *Exam: a comprehensive environment for the analysis of access control policies*, Int. J. Inf. Secur., 9 (2010), pp. 253–273.

[61] D. Lin, P. Rao, E. Bertino, and J. Lobo, *An approach to evaluate policy similarity*, in Proceedings of the 12th ACM symposium on Access control models and technologies, SACMAT '07, New York, NY, USA, 2007, pp. 1–10.

[62] J. Lobo, R. Bhatia, and S. Naqvi, *A policy description language*, in Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference innovative applications of artificial intelligence, AAAI '99/IAAI '99, Menlo Park, CA, USA, 1999, American Association for Artificial Intelligence, pp. 291–298.

[63] A. Margara and G. Cugola, *Processing flows of information: from data stream to complex event processing*, in Proceedings of the 5th ACM international conference on Distributed event-based system, DEBS '11, New York, NY, USA, 2011, ACM, pp. 359–360.

[64] S. Mohammad and G. Hirst, *Distributional measures of semantic distance: A survey*, CoRR, abs/1203.1858 (2012).

[65] M. Moyer, M.J.; Abamad, *Generalized role-based access control*, in Proceedings of the The 21st International Conference on Distributed Computing Systems, Washington, DC, USA, 2001, IEEE Computer Society, pp. 391–398.

[66] E. Network and I. S. A. (ENISA), *Inventory of risk management/risk assessment methods*, Sept. 2012.

[67] Q. Ni, E. Bertino, and J. Lobo, *D-algebra for composing access control policy decisions*, in Proceedings of the 4th International Symposium on Information, Computer, and Communications Security, ASIACCS '09, New York, NY, USA, 2009, ACM, pp. 298–309.

[68] OASIS, *Xacml v3.0 administration and delegation profile version 1.0.*

[69] G. K. P. Ashley, S. Hada, *Enterprise privacy authorization language (epal) specification 1.2*, November 2003.

[70] A. Pawling, P. Yan, J. Candia, T. Schoenharl, and G. Madey, *Anomaly detection in streaming sensor data*, in Intelligent Techniques for Warehousing and Mining Sensor Network Data, IGI Global, 2009.

[71] A. Rostin, O. Albrecht, J. Bauckmann, F. Naumann, and U. Leser, *A machine learning approach to foreign key discovery*, in Proceedings of the 12th International Workshop on the Web and Databases (WebDB 2009), 2009.

[72] R. Sandhu, V. Bhamidipati, and Q. Munawer, *The arbac97 model for role-based administration of roles*, ACM Trans. Inf. Syst. Secur., 2 (1999), pp. 105–135.

[73] S. Schwiderski-Grosche and K. Moody, *The spatec composite event language for spatio-temporal reasoning in mobile systems*, in Proceedings of the Third ACM International Conference on Distributed Event-Based Systems, DEBS '09, New York, NY, USA, 2009, ACM, pp. 11:1–11:12.

[74] Security and Privacy Committee (SPC), *Break-glass: An approach to granting emergency access to healthcare systems*. White paper, Joint NEMA/COCIR/JIRA, 2004.

[75] StreamBase, *http://www.streambase.com/*, Sept. 2013.

[76] M. Strembeck and G. Neumann, *An integrated approach to engineer and enforce context constraints in rbac environments*, ACM Trans. Inf. Syst. Secur., 7 (2004), pp. 392–427.

[77] Y. Thakran and D. Toshniwal, *Unsupervised outlier detection in streaming data using weighted clustering*, in Intelligent Systems Design and Applications (ISDA), 2012 12th International Conference on, 2012, pp. 947–952.

[78] WordNet, *http://wordnet.princeton.edu/*, Mar. 2013.

[79] Z. Wu and M. Palmer, *Verbs semantics and lexical selection*, in Proceedings of the 32nd annual meeting on Association for Computational Linguistics, ACL '94, Stroudsburg, PA, USA, 1994, pp. 133–138.

[80] M. Xu and D. Wijesekera, *A role-based xacml administration and delegation profile and its enforcement architecture*, in Proceedings of the 2009 ACM workshop on Secure web services, SWS '09, New York, NY, USA, 2009, ACM, pp. 53–60.

[81] M. Xu, D. Wijesekera, X. Zhang, and D. Cooray, *Towards session-aware rbac administration and enforcement with xacml*, in Policies for Distributed Systems and Networks, 2009. POLICY 2009. IEEE International Symposium on, july 2009, pp. 9 –16.

[82] Y. Zhang, N. Meratnia, and P. Havinga, *Outlier detection techniques for wireless sensor networks: A survey*, Commun. Surveys Tuts., 12 (2010), pp. 159–170.